Ecole Polytechnique Fédérale de Lausanne

PROGRAMMES ET OBJETS INFORMATIQUES

Prof. Ch. Rapin

Juillet 1989

Laboratoire de compilation

Département d'Informatique
CH-Lausanne (Suisse)

Ecole Polytechnique Fédérale de Lausanne

PROGRAMMES ET OBJETS INFORMATIQUES

Prof. Ch. Rapin

Juillet 1989

Laboratoire de compilation

Département d'Informatique
CH-Lausanne (Suisse)

REPRO EPFL - 1993

Table des matières

Introduction

Dans ce cours, il sera montré comment construire et utiliser différentes structures de données; des applications typiques seront développées. Dans la mesure du possible, il sera adopté une approche orientée objet. Comme support, il sera utilisé le langage Newton; il s'agit d'un langage expérimental, développé et implanté par le Laboratoire de Compilation. Ce cours présuppose la connaissance d'un langage de programmation structuré tel que Pascal; les principaux concepts du langage Newton seront sommairement introduits selon les besoins: seules les notions les plus originales feront l'objet d'une description plus détaillée.

Chapitre 1

Introduction à Newton

Comme la plupart des langages de programmation modernes, Newton permet de manipuler les données arithmétiques, logiques et textuelles au moyen d'un certain nombre de types de données prédéfinis (*integer*, *real*, *Boolean*, *character*, *string*, *alphabet*) et de définir, au moyen de constructeurs appropriés, de nouveaux types de données selon les besoins de chaque application spécifique.

Comme premier exemple, on va partir du programme *booleens* suivant; ce listage peut correspondre au texte tel qu'il aura été préparé à l'éditeur.

```
/* /*newsource=user2:[rapin]booleens*/ */
program booleens declare
(*Ce programme illustre la semantique du type predefini Boolean .*)

 boolean variable p,q:=true
do(*booleens*)
  print("1. Valeurs du type  Boolean:",line,
        "   ========================" ,line,
        line,true,
        line,false,line);
  line;
  print("2. Operateur de negation:",line,
        "   =====================" ,line);
  print(line,column(6),"p",column(20),"~p",line);
  until
    print(line,p,column(16),~p)
  take (p:=~p) repetition;
  line; line;
  print("3. Operateurs conjonctifs:",line,
        "   ======================" ,line);
  print(line,column(6),"p",column(16),"q",
             column(29),"p/\q",column(36),"p NAND q",line);
  until
    until
      print(line,p,column(11),q,column(26),p/\q,column(36),p nand q)
    take (q:=~q) repetition
  take (p:=~p) repetition;
  line; line;
  print("4. Operateurs disjonctifs:",line,
        "   ======================" ,line);
  print(line,column(6),"p",column(16),"q",
             column(29),"p\/q",column(37),"p NOR q",line);
  until
    until
      print(line,p,column(11),q,column(26),p\/q,column(36),p nor q)
    take (q:=~q) repetition
  take (p:=~p) repetition;
  line; line;
  print("5. Operateurs implicatifs:",line,
        "   =====================" ,line);
  print(line,column(6),"p",column(16),"q",
             column(29),"p|>q",column(39),"p<|q",
             column(47),"p EXCL q",column(57),"p NEGT q",line);
```

```
  until
    until
      print(line,p,column(11),q,column(26),p|>q,column(36),p<|q,
            column(45),p excl q,column(55),p negt q)
    take (q:=~q) repetition
  take (p:=~p) repetition;
  line; line;
  print("6. Operateurs d'equivalence:",line,
        "   =========================",line);
  print(line,column(6),"p",column(16),"q",
            column(29),"p==q",column(38),"p~==q",line);
  until
    until
      print(line,p,column(11),q,column(26),p==q,column(36),p~==q)
    take (q:=~q) repetition
  take (p:=~p) repetition;
  line; print("<<<FIN>>>")
done(*booleens*)
```

La première ligne est un <u>pragmat</u>; ces derniers ont la forme de commentaires imbriqués. Lors de la compilation de ce programme, le compilateur va le mettre en forme sur le ficher indiqué après le pragmat *newsource=*; après compilation de cette version mise en forme, on obtient le listage suivant:

```
booleens                                        Vax Newton Compiler 0.2
Page    1
                   Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]BOOLEENS.NEW*/ */
     1 PROGRAM booleens DECLARE
     4 (*Ce programme illustre la semantique du type predefini  Boolean .*)
     4
     4   Boolean VARIABLE p,q:=TRUE
    11 DO(*booleens*)
    12   print("1. Valeurs du type  Boolean:",line,
    18         "   ========================" ,line,
    22         line,TRUE,
    26         line,FALSE,line);
    33   line;
    35   print("2. Operateur de negation:",line,
    41         "   ======================",line);
    46   print(line,column(6),"p",column(20),"~p",line);
    67   UNTIL
    68     print(line,p,column(16),~p)
    82   TAKE (p:=~p) REPETITION;
    91   line; line;
    95   print("3. Operateurs conjonctifs:",line,
   101         "   ======================" ,line);
   106   print(line,column(6),"p",column(16),"q",
   124              column(29),"p/\q",column(36),"p NAND q",line);
   141   UNTIL
   142     UNTIL
   143       print(line,p,column(11),q,column(26),p/\q,column(36),p NAND q)
   174     TAKE (q:=~q) REPETITION
   182   TAKE (p:=~p) REPETITION;
   191   line; line;
```

```
195    print("4. Operateurs disjonctifs:",line,
201          "    ======================" ,line);
206    print(line,column(6),"p",column(16),"q",
224               column(29),"p\/q",column(37),"p NOR q",line);
241    UNTIL
242      UNTIL
243        print(line,p,column(11),q,column(26),p\/q,column(36),p NOR q)
274      TAKE (q:=~q) REPETITION
282    TAKE (p:=~p) REPETITION;
291    line; line;
295    print("5. Operateurs implicatifs:",line,
301          "    ======================" ,line);
306    print(line,column(6),"p",column(16),"q",
324               column(29),"p|>q",column(39),"p<|q",
338               column(47),"p EXCL q",column(57),"p NEGT q",line);
355    UNTIL
356      UNTIL
357        print(line,p,column(11),q,column(26),p|>q,column(36),p<|q,
388              column(45),p EXCL q,column(55),p NEGT q)
406      TAKE (q:=~q) REPETITION
414    TAKE (p:=~p) REPETITION;
423    line; line;
427    print("6. Operateurs d'equivalence:",line,
433          "    ========================" ,line);
438    print(line,column(6),"p",column(16),"q",
456               column(29),"p==q",column(38),"p~==q",line);
473    UNTIL
474      UNTIL
475        print(line,p,column(11),q,column(26),p==q,column(36),p~==q)
506      TAKE (q:=~q) REPETITION
514    TAKE (p:=~p) REPETITION;
523    line; print("<<<FIN>>>")
529 DONE(*booleens*)

   ****  No messages were issued  ****
```

Dans cette mise en forme, les mots clés (réservés) sont mis en majuscules (**program, declare, variable** ...); les identificateurs sont mis en minuscules (à part *Boolean*). Les mots apparaissant dans des chaînes de caractères (encadrées de guillemets ou de dièzes) ou dans des commentaires (encadrés des paires de symboles *(* ... *)* ou /* ... */) ne sont pas modifiés.

Ce programme donne la signification des différentes opérations définies sur les valeurs logiques. On peut constater, d'après les résultats qui suivent que des opérateurs sont disponibles pour toutes les opérations portant sur une ou deux valeurs logiques.

```
1. Valeurs du type  boolean:
   =======================

  'TRUE'
 'FALSE'

2. Operateur de negation:
   ====================

     p              ~p

  'TRUE'         'FALSE'
 'FALSE'          'TRUE'

3. Operateurs conjonctifs:
   =====================

     p         q          p/\q    p NAND q

  'TRUE'    'TRUE'       'TRUE'   'FALSE'
  'TRUE'   'FALSE'      'FALSE'    'TRUE'
 'FALSE'    'TRUE'      'FALSE'    'TRUE'
 'FALSE'   'FALSE'      'FALSE'    'TRUE'

4. Operateurs disjonctifs:
   =====================

     p         q          p\/q     p NOR q

  'TRUE'    'TRUE'       'TRUE'   'FALSE'
  'TRUE'   'FALSE'       'TRUE'   'FALSE'
 'FALSE'    'TRUE'       'TRUE'   'FALSE'
 'FALSE'   'FALSE'      'FALSE'    'TRUE'

5. Operateurs implicatifs:
   =====================

     p         q          p|>q     p<|q    p EXCL q  p NEGT q

  'TRUE'    'TRUE'       'TRUE'   'TRUE'   'FALSE'   'FALSE'
  'TRUE'   'FALSE'      'FALSE'   'TRUE'    'TRUE'   'FALSE'
 'FALSE'    'TRUE'       'TRUE'  'FALSE'   'FALSE'    'TRUE'
 'FALSE'   'FALSE'       'TRUE'   'TRUE'   'FALSE'   'FALSE'

6. Operateurs d'equivalence:
   =======================

     p         q          p==q     p~==q

  'TRUE'    'TRUE'       'TRUE'   'FALSE'
  'TRUE'   'FALSE'      'FALSE'    'TRUE'
 'FALSE'    'TRUE'      'FALSE'    'TRUE'
 'FALSE'   'FALSE'       'TRUE'   'FALSE'
<<<FIN>>>
```

Les différents groupes d'opérations indiquent l'ordre de leur prise en charge; ainsi, en l'absence de parenthésage, la négation sera effectuée avant les opérations conjonctives; les opérations conjonctives le seront avant les opérations disjointives et ainsi de suite.

Remarque:

Pour comparer si deux valeurs logiques sont égales ou différentes, il ne faut jamais utiliser l'opérateur d'égalité = ou son inverse ~= , mais l'opérateur d'équivalence logique == ou son inverse ~== .

Un programme Newton a la structure générale suivante:

program identificateur **declare**
suite_de_déclarations
do suite_d_énoncés **done**

Les déclarations et énoncés successifs sont normalement séparés par des points-virgules; il est possible, mais non obligatoire, de faire suivre la dernière déclaration ou le dernier énoncé d'un point-virgule (ce dernier est alors suivi d'une déclaration, respectivement d'un énoncé vide).

On remarque la forme des déclarations de variables:

indication_de_type **variable** liste_de_variables := expression

L'expression est évaluée au moment de l'enregistrement (l'élaboration) de la déclaration; sa valeur est stockée dans toutes les variables de la liste précédente; cette possibilité d'initialiser les variables au moment de leur déclaration est facultative.

Ce programme comporte plusieurs boucles de la forme:

until
suite_d_énoncé
take *(*assignation*)* **repetition**

Il y figure un énoncé d'assignation à un endroit où on s'attendrait plutôt à trouver une expression. En fait, Newton est un langage d'expression: ceci signifie que la plupart des énoncés peuvent, dans le contexte approprié, produire un résultat qui peut être une valeur ou, dans certains cas, une variable. Ainsi, le résultat d'une assignation est la valeur stockée dans la variable à gauche du symbole := ; lorsqu'on veut ainsi utiliser la valeur produite par une assignation, il est presque toujours nécessaire de parenthéser cette dernière. Ainsi, si *x* et *y* sont des variables réelles, l'énoncé *x:= (y:= 5.7)* stockera la valeur *5.7* dans chacune d'entre elles. Par contre, il sera incorrect d'écrire *x := y := 5.7*; en effet, dans ce dernier cas, les deux assignations seraient interprétées de gauche à droite: le résultat de la première assignation serait la valeur *5.7* stockée en *x* et non la variable *y*; cette valeur n'a donc aucun sens à gauche du deuxième := .

Si la variable logique *p* a initialement la valeur **true**, on peut facilement constater qu'une boucle de la forme ci-après fera exécuter deux fois la séquence d'énoncés incorporée dans cette dernière, la première fois avec *p* == **true** et la seconde avec *p* == **false**. Une fois la boucle quittée *p* a de nouveau la valeur **true**.

until
suite_d_énoncés
take *(p := ~p)* **repetition**

Dans une telle boucle, le mot clé **until** peut être remplacé par **while**, ce qui inverse la condition de sortie. On peut citer également d'autres formes de boucles courantes.

until expression_logique **repeat**
suite_d'énoncés
repetition

Cette fois la condition est testée à l'entrée, et non plus à la sortie de la boucle. On peut combiner les deux formes si la condition doit être testée au milieu de la boucle.

until
suite_d_énoncés
take expression_logique **repeat**
suite_d_énoncés
repetition

Dans ces deux derniers cas, il est évidemment aussi possible de remplacer **until** par **while**. L'énoncé d'assignation traditionnel n'est pas le seul moyen, en Newton, de modifier l'état de variables. Il existe, en particulier, un énoncé d'assignation inverse de la forme:

espression =: variable

La valeur de l'expression est stockée dans la variable à droite du symbole =: ; utilisé dans un expression, cet énoncé livre l'ancienne valeur de la variable concernée. Comme le montre le programme *fibonacci* suivant, cet énoncé permet de simplifier la programmation de certaines relations de recurrence.

```
/* /*OLDSOURCE=USER2:[RAPIN]FIBO.NEW*/ */
PROGRAM fibonacci DECLARE
  CONSTANT limite=40;
  (*Evaluation des nombres de fibonacci pour les valeurs entieres
    non negatives ne depassant pas  limite .
  *)
  integer FUNCTION fib(integer VALUE n)DECLARE
    integer VARIABLE av_der:=1(*fib(-1)*),
                     der   :=0(*fib(0)*)
  DO(*fib*)
    FOR integer VALUE k FROM 1 BY 1 TO n REPEAT
      der+av_der=:der=:av_der
    REPETITION
  TAKE der DONE(*fib*)
DO(*fibonacci*)
  FOR integer VALUE k FROM 0 TO limite REPEAT
    line; edit(k,2,0); edit(fib(k),10,0)
  REPETITION;
  print(line,"<<<FIN>>>")
DONE(*fibonacci*)
```

Ce programme évalue les nombres de Fibonacci définis par les relations suivantes:

$fib\,(0) = 0$
$fib\,(1) = 1$
$fib\,(n) = fib\,(n-1) + fib\,(n-2)$ *pour* $n \geq 2$

```
0          0
1          1
2          1
3          2
4          3
5          5
6          8
7         13
8         21
9         34
10        55
11        89
12       144
13       233
14       377
15       610
16       987
17      1597
18      2584
19      4181
20      6765
21     10946
22     17711
23     28657
24     46368
25     75025
26    121393
27    196418
28    317811
29    514229
30    832040
31   1346269
32   2178309
33   3524578
34   5702887
35   9227465
36  14930352
37  24157817
38  39088169
39  63245986
40 102334155
<<<FIN>>>
```

Ce programme montre la manière dont une **fonction** est déclarée:

> indication_de_type **function** identificateur
>   paramètres_formels
> **declare**
>   suite_de_déclarations
> **do**
>   suite_d_énoncés
> **take** expression **done**

L'expression entre les symboles **take** et **done** livre la valeur de l'application de fonction concernée. Dans la fonction *fib*, on remarque que la variable *av_der* est initialisée à la valeur fictive *fib (-1)*, ce qui évite de devoir traiter spécialement le cas *n = 0*.
La boucle **for** de la fonction *fib* est exécutée pour *k* variant de *1* à n par pas de *1*; elle n'est pas exécutée si *n < 1*.

Par contraste, la boucle **for** de la partie executable du programme est toujours elaborée au moins une fois. En-effet, vu l'absence d'un pas de tabulation (introduit par le symbole **by**) cette boucle sera exécutée pour *k* variant de *0* à *limite* avec un pas égal à *1* si *limite > 0* et à *-1* si *limite < 0*.

Dans la boucle **for** de la fonction, on remarque que la double assignation inverse *der + av_der =: der =: av_der* permet d'éviter le recours à une variable auxiliaire; elle simplifie de plus la formulation. En son absence, on aurait une séquence du genre *nouv := der + av_der; av_der := der; der := nouv*.
Vu que le compteur *k* n'est pas utilisé à l'intérieur de la boucle, la clause **value** *k* de l'énoncé **for** peut être omise.

Pour des raisons analogues, on dispose aussi <u>d'un énoncé d'échange du contenu de variables</u>:

> variable *:=:* variable

Les contenus de deux variables concernées sont échangés; si l'une des variables n'est pas initialisée avant cette opération, l'autre ne le sera pas après. Utilisé dans une expression, cet énoncé produit pour résultat la variable à droite du symbole *:=:* (<u>attention</u>: c'est bien la variable elle-même qui est livrée comme résultat et non uniquement son contenu). Utilisé de manière répétitive, cet énoncé permet de permuter circulairement le contenu d'un nombre arbitraire de variables du même type. Ceci apparait clairement dans le programme *varot* suivant:

```
varot                                                   Vax Newton Compiler 0.2
Page    1
                       Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]VAROT.NEW*/ */
     1 PROGRAM varot DECLARE
     4   real VARIABLE x1:=1.111,x2:=2.222,x3:=3.333,x4:=4.444,x5:=5.555
    35 DO(*varot*)
    36   line;
    38   edit(x1,7,3); edit(x2,7,3); edit(x3,7,3); edit(x4,7,3); edit(x5,7,3);
    83   x1:=:x2:=:x3:=:x4:=:x5;
    93   line;
    95   edit(x1,7,3); edit(x2,7,3); edit(x3,7,3); edit(x4,7,3); edit(x5,7,3);
   140   line; print("<<<FIN>>>")
   146 DONE(*varot*)

         **** No messages were issued ****
```

Résultats

```
 1.111  2.222  3.333  4.444  5.555
 2.222  3.333  4.444  5.555  1.111
<<<FIN>>>
```

Le quadruple échange *x 1 :=: x 2 :=: x 3 :=: x 4 :=: x 5* a permuté circulairement les contenus des cinq variables *x 1 , x 2, x 3, x 4 et x 5*. En l'absence de cet énoncé, il faudrait de nouveau avoir recours à une variable supplémentaire et à une séquence plus compliquée du genre *temp := x 1; x 1 := x 2, x 2 := x 3; x 3 := x 4 := x 5; x 5 := temp*.

En s'appuyant sur la notion de <u>référence</u> (ou repère), le langage Newton permet de manipuler les variables en tant que telles (indépendemment de la valeur de leur contenu). Au moyen d'un <u>énoncé de dénotation</u>, il est possible d'établir un accès indirect à une variable par l'intermédiaire d'une référence. Cet énoncé a la forme:

reference → variable

Utilisé comme expression, une dénotation livre pour résultat la variable située à droite du symbole →. Le programme *references* suivant illustre ce concept.

references Vax Newton Compiler 0.2
Page 1

Source listing

```
/* /*OLDSOURCE=USER2:[RAPIN]REFS.NEW*/ */
PROGRAM references DECLARE
  real VARIABLE x:=1.111,y:=2.222,z:=3.333;
  real REFERENCE xx->x,yy->y,zz->z;

  PROCEDURE impr DO
    print(line,"***Contenu des variables***",
          line,"x =",edit(x,7,3),
            __,"y =",edit(y,7,3),
            __,"z =",edit(z,7,3),
          line,"xx=",edit(xx,7,3),
            __,"yy=",edit(yy,7,3),
            __,"zz=",edit(zz,7,3));
    print(line,"***Comparaison d'egalite des variables***",
          line,xx=x,xx=y,xx=z,
          line,yy=x,yy=y,yy=z,
          line,zz=x,zz=y,zz=z);
    print(line,"***Comparaison d'identite des variables***",
          line,xx IS x,xx IS y,xx IS z,
          line,yy IS x,yy IS y,yy IS z,
          line,zz IS x,zz IS y,zz IS z,line)
  DONE(*impr*)
DO(*references*)
  print(line,"*****Etat initial*****"); impr;
  yy->z:=x;
  print(line,"*****Deuxieme etat*****"); impr;
  zz->x:=y;
  print(line,"*****Troisieme etat*****"); impr;
  xx->y:=z;
  print(line,"*****Etat final*****"); impr;
  print(line,"<<<FIN>>>")
DONE(*references*)
```

**** No messages were issued ****

Résultats

```
*****Etat initial*****
***Contenu des variables***
x =  1.111  y =  2.222  z =  3.333
xx=  1.111  yy=  2.222  zz=  3.333
***Comparaison d'egalite des variables***
  'TRUE' 'FALSE' 'FALSE'
 'FALSE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'
***Comparaison d'identite des variables***
  'TRUE' 'FALSE' 'FALSE'
 'FALSE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'

*****Deuxieme etat*****
***Contenu des variables***
x =  1.111  y =  2.222  z =  1.111
xx=  1.111  yy=  1.111  zz=  1.111
***Comparaison d'egalite des variables***
  'TRUE' 'FALSE'  'TRUE'
  'TRUE' 'FALSE'  'TRUE'
  'TRUE' 'FALSE'  'TRUE'
***Comparaison d'identite des variables***
  'TRUE' 'FALSE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'
 'FALSE' 'FALSE'  'TRUE'

*****Troisieme etat*****
***Contenu des variables***
x =  2.222  y =  2.222  z =  1.111
xx=  2.222  yy=  1.111  zz=  2.222
***Comparaison d'egalite des variables***
  'TRUE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'
  'TRUE'  'TRUE' 'FALSE'
***Comparaison d'identite des variables***
  'TRUE' 'FALSE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'
  'TRUE' 'FALSE' 'FALSE'

*****Etat final*****
***Contenu des variables***
x =  2.222  y =  1.111  z =  1.111
xx=  1.111  yy=  1.111  zz=  2.222
***Comparaison d'egalite des variables***
 'FALSE'  'TRUE'  'TRUE'
 'FALSE'  'TRUE'  'TRUE'
  'TRUE' 'FALSE' 'FALSE'
***Comparaison d'identite des variables***
 'FALSE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'
  'TRUE' 'FALSE' 'FALSE'

<<<FIN>>>
```

Pour comprendre ces résultats, on peut faire une série de figures illustrant l'état successif du système. Après l'élaboration de la partie déclarative du programme, on a la situation de la figure 1.

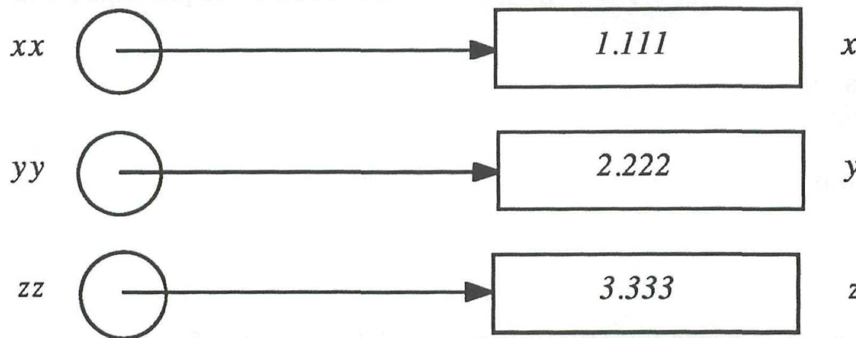


Figure 1

Les références *xx*, *yy* et *zz* ont été déclarées au moyen d'une déclaration de structure analogue à celle des variables:

indication_de_type **refence** liste_de_références → variable

La dénotation initiale (→ variable) est facultative; de plus, comme pour les variables, plusieurs groupes de références peuvent être déclarés et initialisés au moyen de la même déclaration.

Le double énoncé *yy* → *z* := *x* va tout d'abord faire porter la référence *yy* sur la variable *z*; cette dernière est le résultat de l'opération. A cette variable, il est ensuite stocké le contenu *1.111* de la variable *x*; on trouve alors l'état de la figure 2.

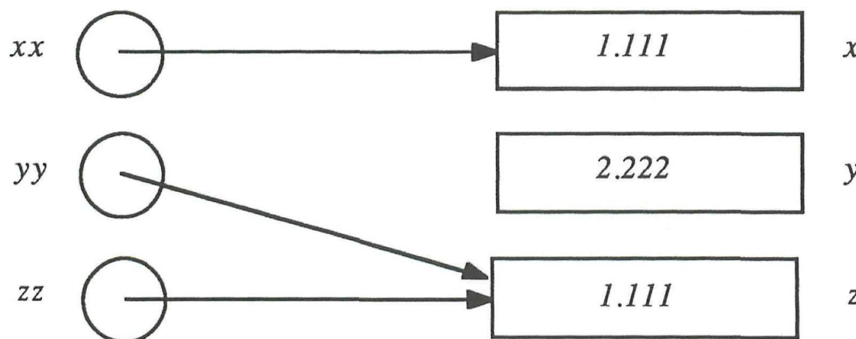


Figure 2

Cette figure permet d'interpréter la deuxième partie des résultats. En plus de la comparaison d'égalité de deux valeurs (notée au moyen de l'opérateur =), il est possible (au moyen de l'opérateur **is**) de comparer l'identité de deux variables. Ainsi dans l'état de la figure 2, on remarque que la relation *xx* = *z* est vraie puisque le contenu de la variable *x* repérée par la référence *xx* est égal à celui de la variable *z*; par contre, la relation *xx* **is** *z* est fausse puisque la variable *x* repérée par *xx* est distincte de *z*.

Ce programme fait intervenir une procédure *impr*; en général, la déclaration d'une procédure a la forme:

```
procedure identificateur
    paramètres_formels
declare
    suite_de_déclarations
do
    suite_d_énoncés
done
```

Comme l'indique la procédure *impr*, la partie formelle et la partie déclarative sont facultatives.

Chapitre 2

La notion d'objet

Par définition, un objet est un paquet d'informations manipulable tant que tel. Les objets les plus simples ne contiennent que des données passives. Le programme *objets* suivant en montre un exemple.

```
objets                                          Vax Newton Compiler 0.2
Page    1
                    Source listing

/* /*OLDSOURCE=USER2:[RAPIN]OBJETS.NEW*/ */
PROGRAM objets DECLARE
  OBJECT nom_var(string VALUE nom; integer VARIABLE var);

  nom_var VALUE nv1=nom_var("Toto",111),
                nv2=nom_var("Toto",222),
                nv3=nv2,
                nv4=nom_var("Titi",444);

  PROCEDURE impr_nom_var(nom_var VALUE nv)DO
    print(___,nv.nom); edit(nv.var,5,0)
  DONE(*impr_nom_var*);

  Boolean FUNCTION equiv
    (nom_var VALUE nv1,nv2)
  DO TAKE nv1.nom=nv2.nom/\nv1.var=nv2.var DONE;

  PROCEDURE traite_objets
    (nom_var VARIABLE nv; PROCEDURE acte)
  DO(*traite_objets*)
    nv:=nv1; acte;
    nv:=nv2; acte;
    nv:=nv3; acte;
    nv:=nv4; acte
  DONE(*traite_objets*);

  PROCEDURE imprime_etat DO
    print(line,"***Etat des objets***",line);
    traite_objets(LAMBDA VALUE obj,impr_nom_var(obj));
    print(line,"***Comparaison d'egalite des objets***",line);
    traite_objets(LAMBDA VALUE obj_gauche,
                  (traite_objets(LAMBDA VALUE obj_droite,
                                 print(obj_gauche=obj_droite));
                   line));
    print("***Comparaison d'equivalence des objets***",line);
    traite_objets(LAMBDA VALUE obj_gauche,
                  (traite_objets(LAMBDA VALUE obj_droite,
                                 print(equiv(obj_gauche,obj_droite)));
                   line));
    print("<<<FIN>>>",line)
  DONE(*imprime_etat*)
DO(*objets*)
  print("*****Etat initial*****",line); imprime_etat;
  nv2.var:=111;
  print(line,"*****Etat final*****",line); imprime_etat
DONE(*objets*)

   ****  No messages were issued  ****
```

On constate, tout d'abord, la déclaration d'objet de la forme suivante:

object identificateur paramètres_formels

Dans le cas présent, *nom_var* désigne un nouveau type d'objet. Chaque objet de ce type comporte une chaîne de caractères *nom* immuable pendant toute la durée de vie de l'objet et une variable entière *var*. Un nouvel objet est créé au moyen d'un générateur d'objet de la forme:

type_ objet paramètre_effectifs

Ainsi, dans le programme *objets*, le générateur d'objet *nom_var ("Toto", 111)* a pour effet d'allouer, en mémoire, un nouvel objet du type *nom_var*. Le paramètre (la composante) *nom* de cet objet a pour valeur la chaîne *"Toto"*; sa composante *var* est initialisée par la valeur *111*. D'une manière générale, les paramètres effectifs d'un générateur d'objet ont pour rôle d'initialiser les composantes de l'objet dénotées par les paramètres formels correspondants; la correspondance entre les paramètres effectifs d'un générateur d'objet et les paramètres formels du type objet correspondant est établie selon les mêmes règles (précisées plus loin) que dans le cas d'applications de procédures ou de fonctions.

Ce programme *objets* construit trois objets *nv1*, *nv2* et *nv4* du type *nom_var*; par contre *nv3* désigne le même objet que *nv2*. L'état initial du système est représenté dans la figure 3:

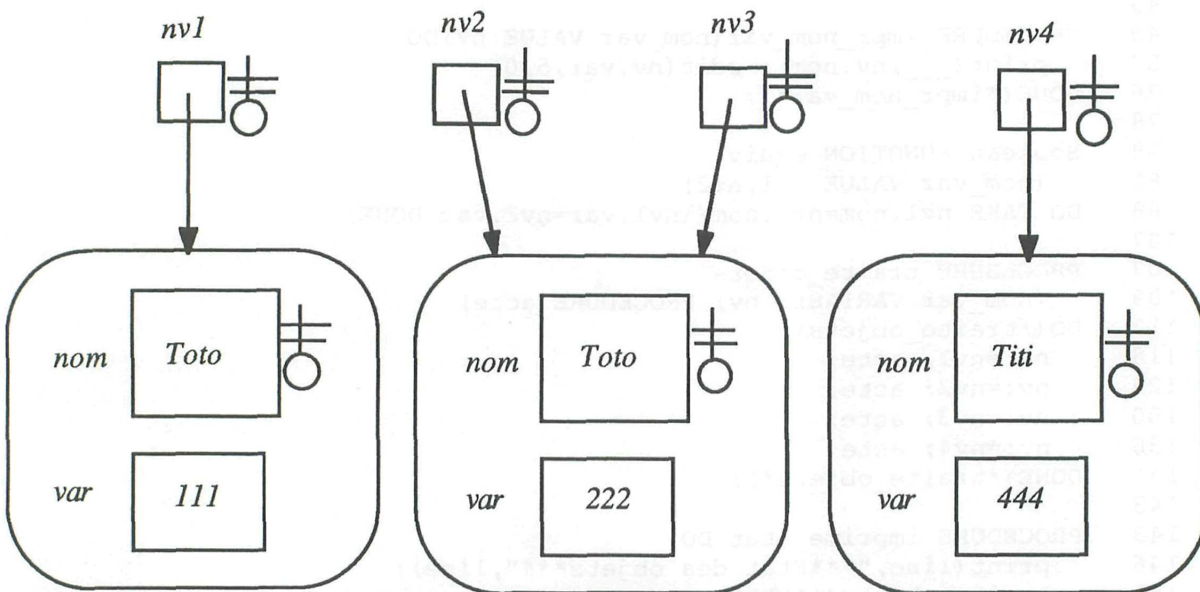


Figure 3

Les cadenas symboliques à côté des entités *nv1*, *nv2*, *nv3*, *nv4* et des composantes *nom* signifient que les valeurs correspondantes ne peuvent pas être changées au moyen d'un énoncé d'assignation. Ainsi, pendant toute l'exécution du programme, *nv1* désignera toujours le même objet; la composante nom de ce dernier sera toujours égale à *"Toto"*.
D'une manière générale, il est possible de déclarer des valeurs au moyen de déclarations de la forme:

indication_de_type **value** identificateur = expression

L'identificateur sur lequel porte une telle déclaration de valeur dénote, pendant toute la durée de vie du bloc correspondant, la valeur de l'expression à droite du symbole d'égalité: cette expression est évaluée une fois pour toutes lors de l'élaboration de la déclaration.

Les résultats de l'exécution du programme *objets* sont les suivants:

```
*****Etat initial*****

***Etat des objets***
   Toto 111   Toto 222   Toto 222   Titi 444
***Comparaison d'egalite des objets***
  'TRUE' 'FALSE' 'FALSE' 'FALSE'
 'FALSE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE' 'FALSE'  'TRUE'
***Comparaison d'equivalence des objets***
  'TRUE' 'FALSE' 'FALSE' 'FALSE'
 'FALSE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE' 'FALSE'  'TRUE'
<<<FIN>>>

*****Etat final*****

***Etat des objets***
   Toto 111   Toto 111   Toto 111   Titi 444
***Comparaison d'egalite des objets***
  'TRUE' 'FALSE' 'FALSE' 'FALSE'
 'FALSE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE' 'FALSE'  'TRUE'
***Comparaison d'equivalence des objets***
  'TRUE'  'TRUE'  'TRUE' 'FALSE'
  'TRUE'  'TRUE'  'TRUE' 'FALSE'
  'TRUE'  'TRUE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE' 'FALSE'  'TRUE'
<<<FIN>>>
```

La première moitié des résultats reflète la situation de la figure 3. On a comparé entre eux les objets *nv1*, *nv2*, *nv3* et *nv4* au moyen de l'opérateur d'égalité; on remarque notamment que l'on a *nv2* = *nv3* puisqu'il s'agit du même objet. On a fait une deuxième série de comparaisons au moyen de la fonction logique *equiv*: celle-ci est vraie si les composantes homologues des deux objets concernés ont les mêmes valeurs. Dans l'état initial, aucun des trois objets n'est équivalent à un autre.

Après l'impression de l'état initial, il est fait l'assignation *nv2 . var := 111*; ceci a pour effet de changer la valeur de la composante *var* de l'objet *nv2*. En inspectant la deuxième moitié des résultats, on peut constater les éléments suivants:

- Le changement de valeur de la composante *var* apparaît non seulement pour l'objet *nv2* mais également pour *nv3* (puisqu'il s'agit du même objet).

- L'objet *nv2* (ou *nv3*) ainsi modifié devient équivalent à *nv1* puisque leurs composantes homologues sont maintenant égales.

- Par contre, lors de l'application de la comparaison d'égalité, l'objet *nv2* reste, bien entendu, distinct de *nv1*.

A plusieurs reprises, dans ce programme, il a été nécessaire d'accomplir une action donnée pour les quatre valeurs *nv1*, *nv2*, *nv3* et *nv4*. Chaque fois, la chose a été accomplie par l'intermédiaire de la procédure *traite_objets*. Pour saisir son fonctionnement, il est nécessaire d'examiner de plus près les différentes manières de communiquer des paramètres à une procédure; cette procédure fait intervenir des passages de paramètres par nom (les paramètres effectifs associés au paramètre formel *acte*), ainsi que des des paramètres effectifs liés (les

paramètres effectifs associés au paramètre formel *nv*). Une procédure formelle est un paramètre formel introduit au moyen d'une spécification de la forme:

procedure identificateur

Comme paramètre effectif, il doit être associé à une procédure formelle un énoncé; si nécessaire, cet énoncé peut avoir la forme d'une suite parenthésée d'énoncés: les énoncés individuels sont alors séparés par des points virgules. Les énoncés d'assignation, d'assignation inverse, ... doivent, eux-aussi, être parenthésés lorsqu'ils sont associés à une procédure formelle.

Le paramètre effectif associé à une procédure formelle n'est pas évalué au moment d'entrer dans la procédure (ou fonction) contenant cette dernière: il est, par contre exécuté à chaque utilisation du paramètre formel dans le corps de la procédure.

Bien que la chose n'apparaisse pas dans ce programme il est possible d'avoir des expressions formelles qui seront traitées de manière analogue. Une expression formelle est introduite au moyen d'une spécification de la forme suivante dans la partie formelle d'une procédure, fonction ou déclaration d'objet:

indication_de_type **expression** identificateur

Le paramètre effectif associé à une expression formelle sera une expression du type approprié (ou convertible dans ce type), voire une énoncé utilisé comme expression (si cet énoncé est à même de produire une valeur du type approprié). Dans ce cas aussi, le paramètre effectif associé à une expression formelle n'est pas évalué au moment de l'entrée dans la procédure (ou fonction) contenant le paramètre, mais lors de chaque utilisation de ce dernier dans la procédure. Par définition, on dira que le paramètre effectif associé à une procédure ou à une expression formelle lui est transmis par nom.

Le plus souvent, un paramètre formel est une valeur, une variable ou une référence locale à la procédure, la fonction ou l'objet correspondant. Ces trois cas correspondent à une spécification des formes respectives suivantes dans la partie formelle de la procédure, de la fonction ou du type objet.

indication_de_type **value** identificateur
indication_de_type **variable** identificateur
indication_de_type **reference** identificateur

Syntaxiquement, ces spécifications ressemblent à des déclarations de valeur, de variable ou de référence sauf qu'elles ne comportent jamais d'initialisation; l'initialisation des entités correspondantes intervient par l'intermédiaire des paramètres effectifs qui leur seront associés. Par définition, le paramètre effectif associé à un paramètre formel spécifié par **value** lui est transmis par valeur constante. De même, le paramètre effectif associé à un paramètre formel spécifié par **variable** lui est transmis par valeur. Finalement, le paramètre effectif associé à un paramètre formel spécifié par **reference** lui est transmis par référence.

Remarque:

Un paramètre formel spécifié au moyen du symbole **variable** n'est pas équivalent à un paramètre **var** du langage Pascal! Un tel paramètre est une variable locale à la procédure correspondante, initialisée par la valeur du paramètre effectif. L'effet des paramètres **var** de Pascal est obtenu au moyen du passage par référence; en-effet, un paramètre **var** de Pascal n'est pas vraiment une variable, mais bien une référence à la variable qui lui est communiquée comme paramètre effectif.

Le paramètre effectif associé à un paramètre formel spécifié comme valeur, comme variable ou comme référence peut être libre ou lié. La forme libre est celle que l'on trouve dans

pratiquement tous les langages de programmation. On a des possibilités données dans le tableau suivant:

Paramètre formel	Mode de transmission	Paramètre effectif
value	valeur constante	expression
variable	valeur	expression
reference	référence	variable

Un paramètre effectif lié débute par le symbole λ (ou **lambda**); ce symbole est usuellement suivi d'un identificateur; cet identificateur dénote alors la même entité que le paramètre formel: il est utilisable, comme tel, dans toute la partie subséquente de la liste de paramètres effectifs. Les syntaxes possibles des paramètres effectifs liés dépendent du mode de transmission; on a donc trois cas.

Premier cas: Transmission par valeur constante

Un paramètre effectif lié aura la forme suivante:

λ identificateur = expression

Deuxième cas: Transmission par valeur:

λ **value** identificateur := expression

λ **value** identificateur

λ identificateur := expression

λ identificateur

λ

- Dans les formes où l'expression est absente, le paramètre formel n'est pas initialisé.

- Dans les formes où l'identificateur lié est précédé de la spécification λ **value**, seule la valeur du paramètre formel est consultable dans la liste des paramètres effectifs correspondante. En l'absence du symbole **value**, il serait possible de modifier, au moyen d'une assignation, la valeur du paramètre formel depuis la liste des paramètres effectifs. En général, il est préférable d'assurer la protection du paramètre formel avec la spécification λ **value**.

- La forme dégénérée avec le seul symbole λ signifie simplement que le paramètre formel ne sera pas initialisé: il n'y a pas d'identificateur lié dans ce cas.

Troisième cas: Transmission par référence

Un paramètre effectif lié peut apparaître sous l'une des formes suivantes:

> λ identificateur → variable
> λ identificateur
> λ

Là aussi, il y a une forme dégénérée. Dans les formes sans variable, le paramètre formel n'est pas initialisé.

Les paramètres effectifs liés sont les plus souvent utilisés en conjonction avec les paramètres transmis pour nom: c'est l'effet Jensen. C'est cet effet qui est utilisé lors des applications de la procédure *traite_objets*. Ainsi on a la situation représentée dans la figure 4 lors de l'application de procédure *traite_objets* (λ **value** *obj, impr_nom_var(obj))* immédiatement après l'élaboration de l'assignation *nv* := *nv 1*, c'est-à-dire au moment où la procédure formelle *acte* doit être exécutée.

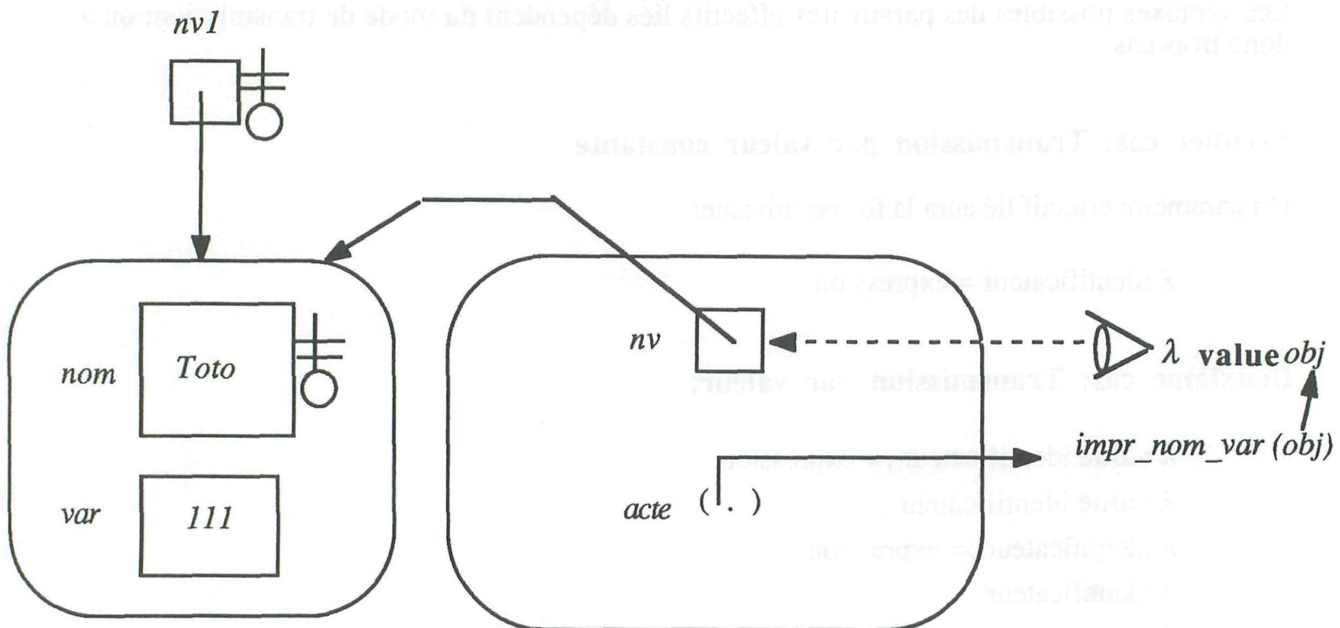


Figure 4

L'objet à droite de la figure est le bloc d'activation de la procédure *traite_objets*, c'est-à-dire la zone de mémoire qui contient les informations locales à cette dernière. L'élaboration de la procédure formelle *acte* reviendra à exécuter l'énoncé *impr_nom_var (obj);* dans ce dernier *obj* est l'identificateur lié au moyen de la clause λ **value** *obj*: sa valeur est celle du paramètre formel *nv*, c'est-à-dire l'objet *nv1*. (Dans la figure, l'oeil symbolique montre qu'il est possible de consulter, au moyen de l'identificateur lié *obj*, le paramètre formel *nv* mais non de modifier sa valeur). Le mécanisme devrait maintenant être clair.

Dans les appels subséquents de la procédure *traite_objets,* on constate la possibilité d'imbriquer ces derniers ce qui est utile pour produire les tables à deux entrées des comparaisons d'égalité et d'équivalence des objets *nv1*, *nv2*, *nv3*, et *nv4*.

<u>D'une manière générale, un objet ne contient pas uniquement des données passives; il inclut en plus les primitives susceptibles de modifier</u>, de manière cohérente, <u>l'état de l'objet</u>.

On suppose, par exemple, que l'on veut modéliser un compte en banque. Un modèle très (trop) simpliste serait de le représenter au moyen d'une variable arithmétique dont le contenu serait égal à la somme disponible dans le compte. Cette manière de faire présente l'inconvénient que n'importe qui pourrait faire n'importe quelle opération sur cette variable. Pour éviter ce genre d'inconvénient, on peut représenter un compte au moyen d'un objet qui modélisera de plus près un compte réel: seules seront autorisées sur cet objet les opérations qui peuvent effectivement être (légalement) réalisées sur un compte réel.
On va en illustrer le principe dans le programme *comptes* suivant. Dans ce programme, on va définir une classe *compte* d'objets; chacune de ces objets modélisera un compte. Lors de la création d'un compte, on doit donner:

- le nom et le prénom du possesseur du compte.

- le numéro du compte.

- la somme qui sera initialement versée dans le compte: cette somme sera au minimum égale à *dépôt_minimum = 50*; dans le cas contraire, le compte est crée, mais il reste fermé (il n'y est rien versé et il n'est pas possible de l'utiliser avant de l'avoir ouvert explicitement avec un dépôt suffisant).

Une fois qu'un compte a été créé et ouvert, il est possible d'y déposer une somme, de consulter son état et d'en retirer une somme; pour cette dernière opération, il est vérifié qu'un montant au moins égal au *dépôt_minimum* subsiste dans le compte: sinon, le retrait n'est pas fait. Chaque opération sur un compte doit être validée; pour cela, il est nécessaire de fournir le numéro du compte: si ce numéro est incorrect, l'opération n'est pas faite.

```
comptes                                              Vax Newton Compiler 0.2
Page    1
                   Source listing

/* /*OLDSOURCE=USER2:[RAPIN]COMPTES.NEW*/ */
PROGRAM comptes DECLARE
  CONSTANT depot_minimum=50;

  integer SUBRANGE naturel
    (naturel>=0
     DEFAULT
       print(line,
             "###Entier naturel negatif invalide remplace par zero###")
     TAKE 0);

  CLASS compte
    ATTRIBUTE nom,prenom,ouvrir,deposer,consulter,retirer
    (string VALUE nom,prenom; naturel VALUE numero_compte;
     naturel VALUE depot_initial)
  (*Un objet de la classe  compte  modelise un compte banquaire.
    En parametre, sont fournis les nom et prenom du possesseur du
    compte, le numero qui lui est attribue ainsi que le depot qui
    y est place initialement; de depot doit au moins etre egal a
    depot_minimum . Une operation sur le compte n'est validee sous
    condition que le numero de compte soit bien celui du possesseur
    du compte.
  *)
  DECLARE(*compte*)
    Boolean VARIABLE ouvert:=FALSE;

    naturel VARIABLE montant:=0;

```

```
PROCEDURE action
  (string VALUE nom_op; naturel VALUE num; PROCEDURE acte)
 (*Effectue l'operation  acte  apres avoir verifie la
   correction du numero de compte.
 *)
DO(*action*)
   print(line,"***",nom_op,_"sur le compte de"_,nom,_,prenom,"***");
   IF num~=numero_compte THEN
      print(line,
               ___"###Numero compte incorrect: operation annulee###") ELSE
   UNLESS ouvert THEN
      print(line,
               ___"###Compte n'est pas ouvert: operation annulee###")
   DEFAULT acte DONE
DONE(*action*);

PROCEDURE deposer(naturel VALUE num,somme)DO
(*Depose le montant  somme  dans le compte apres verification du
  numero  num .
*)
   action("Depot",num,
              print(line,___"Montant depose:"_,somme,"Fr.",
                       line,___"Nouveau credit:"_,
                              (montant:=montant+somme),"Fr."))
DONE(*deposer*);

PROCEDURE consulter(naturel VALUE num)DO
(*Consulte la valeur du credit apres verification du numero  num *)
   action("Consultation",num,
              print(line,___"Credit:"_,montant,"Fr."))
DONE(*consulter*);

PROCEDURE retirer(naturel VALUE somme,num)DO
(*Retire le montant  somme  du compte apres verification du numero
  num . L'operation n'est validee que si le credit residuel est au
  moins egal a  depot_minimum .
*)
   action("Retrait",num,
             IF montant-somme>=depot_minimum THEN
                print(line,___"Montant retire:"_,somme,"Fr.",
                         line,___"Credit residuel:"_,
                              (montant:=montant-somme),"Fr.")
             DEFAULT
                print(line,___"###Retrait de"_,somme,"Fr. impossible###",
                         line,______"Credit de"_,montant,"Fr. insuffisant")
             DONE)
DONE(*retirer*);

PROCEDURE ouvrir(naturel VALUE num,somme)DO
(*Ouverture du compte avec montant initial  somme  apres verifi-
  cation du numero  num . Le compte ne sera pas ouvert si
  somme<depot_minimum .
*)
  print(line,"*****Ouverture du compte de"_,nom,_,prenom,"*****");
  UNLESS ouvert THEN
     IF num=numero_compte THEN
        print(line,_____"Numero de compte:"_,numero_compte);
        IF ouvert:=somme>=depot_minimum THEN
           print(line,_____"Depot initial:"_,(montant:=somme),"Fr.")
        DEFAULT
           print(line,_____"###Depot initial insuffisant,"_
                          "operation annulee###")
        DONE
     DEFAULT
```

```
367            print(line,_____"###Numero de compte incorrect,"_
374                            "operation annulee###")
376          DONE
377        DEFAULT
378          print(line,_____"###Compte deja ouvert,"_
385                          "operation annulee###")
387        DONE
388      DONE(*ouvrir*)
389    DO(*compte*)
390      ouvrir(numero_compte,depot_initial)
396    DONE(*compte*);
398
398    compte VALUE
400      compte_dupont=compte("Dupont","Marcel",102_658_037,10_000),
413      compte_martin=compte("Martin","Julie",366_098_451,50),
426      compte_pinar =compte("Pinar","Srahlec",997_080_777,5),
439      compte_zoom  =compte("Zoom","la Terreur",862_557_143,1_000_000_000)
451  DO(*comptes*)
452    compte_pinar.ouvrir(999_080_777,50);
461    compte_pinar.retirer(45,997_080_777);
470    compte_pinar.ouvrir(997080777,50);
479    compte_pinar.retirer(45,997_080_777);
488    compte_pinar.consulter(997_080_777);
495    compte_dupont.retirer(1_000_000,102_658_037);
504    compte_dupont.retirer(462,102_658_037);
513    compte_zoom.retirer(402_537_000,862_557_143);
522    compte_martin.retirer(4365,366_098_452);
531    compte_martin.retirer(4365,366_098_451);
540    compte_zoom.ouvrir(862_557_143,662_038_157);
549    compte_zoom.deposer(862_557_143,662_038_157);
558    compte_dupont.deposer(3875,102_658_037);
567    compte_dupont.deposer(102_658_037,3875);
576    compte_zoom.consulter(862_557_143);
583    print(line,"<<<FIN>>>")
589  DONE(*comptes*)

     ****  No messages were issued  ****
```

Ce programme fait apparaître plusieurs éléments nouveaux. On constate qu'un compte est modélisé (représenté) au moyen d'un objet du type *compte*. Ces objets sont définis au moyen d'une déclaration de classe; une telle déclaration généralise une déclaration d'objet: elle peut avoir la structure suivante:

> **class** identificateur
>   attributs
>   paramètres_formels
> **declare**
>   suite_de_déclarations
> **do**
>   suite_d_énoncé
> **done**

```
*****Ouverture du compte de Dupont Marcel*****
     Numero de compte:    102658037
     Depot initial:        10000Fr.
*****Ouverture du compte de Martin Julie*****
     Numero de compte:    366098451
     Depot initial:           50Fr.
*****Ouverture du compte de Pinar Srahlec*****
     Numero de compte:    997080777
     ###Depot initial insuffisant, operation annulee###
```

```
*****Ouverture du compte de Zoom la Terreur*****
     Numero de compte:     862557143
     Depot initial:    1000000000Fr.
*****Ouverture du compte de Pinar Srahlec*****
     ###Numero de compte incorrect, operation annulee###
***Retrait sur le compte de Pinar Srahlec***
   ###Compte n'est pas ouvert: operation annulee###
*****Ouverture du compte de Pinar Srahlec*****
     Numero de compte:     997080777
     Depot initial:               50Fr.
***Retrait sur le compte de Pinar Srahlec***
   ###Retrait de              45Fr. impossible###
    Credit de                50Fr. insuffisant
***Consultation sur le compte de Pinar Srahlec***
   Credit:          50Fr.
***Retrait sur le compte de Dupont Marcel***
   ###Retrait de       1000000Fr. impossible###
    Credit de            10000Fr. insuffisant
***Retrait sur le compte de Dupont Marcel***
   Montant retire:             462Fr.
   Credit residuel:           9538Fr.
***Retrait sur le compte de Zoom la Terreur***
   Montant retire:     402537000Fr.
   Credit residuel:     597463000Fr.
***Retrait sur le compte de Martin Julie***
   ###Numero compte incorrect: operation annulee###
***Retrait sur le compte de Martin Julie***
   ###Retrait de            4365Fr. impossible###
    Credit de                50Fr. insuffisant
*****Ouverture du compte de Zoom la Terreur*****
     ###Compte deja ouvert, operation annulee###
***Depot sur le compte de Zoom la Terreur***
   Montant depose:     662038157Fr.
   Nouveau credit:   1259501157Fr.
***Depot sur le compte de Dupont Marcel***
   ###Numero compte incorrect: operation annulee###
***Depot sur le compte de Dupont Marcel***
   Montant depose:            3875Fr.
   Nouveau credit:          13413Fr.
***Consultation sur le compte de Zoom la Terreur***
   Credit:     1259501157Fr.
<<<FIN>>>
```

Il y a une certaine ressemblance entre une déclaration de classe et une déclaration de procédure sauf que la première comporte, en général, des attributs: ces derniers ont la forme d'une liste d'identificateurs séparés par des virgules et procédés du symbole **attribute**. Chaque attribut doit faire l'objet d'une déclaration dans la partie déclarative de la classe ou d'une spécification dans sa partie formelle. Les attributs d'une déclaration de classe dénotent les entités définies dans cette dernière susceptibles d'être utilisées à l'extérieur de la classe pour consulter ou modifier l'état des objets correspondants. On considère, par exemple, la déclaration de valeur:

compte **value** *compte_dupont =*
compte ("Dupont", "Marcel", 102_658_037, 10_000)

Le générateur d'objet à droite du signe d'égalité est d'abord élaboré comme s'il agissait d'un appel de procédure. Après l'exécution de l'algorithme inclus dans la déclaration de classe, le bloc d'activation correspondant (c'est-à-dire la zone de mémoire contenant toutes les entités définies dans la partie formelle et la partie déclarative de la classe) survit: ce bloc forme l'objet produit par le générateur.

Au moyen de la notation pointée, il est possible d'appliquer aux objets individuels d'une classe les attributs de cette dernière. Ainsi, dans l'exemple précédent, on aura *compte_dupont . prénom = "Marcel"*; par contre, des entités telles que *compte_dupont . numéro_compte* ou

compte_dupont . montant ne sont pas définies puisque *numéro_compte* et *montant* ne sont pas des attributs de la classe *compte*. Il n'est en particulier pas possible de modifier brutalement la variable *montant* d'un compte au moyen d'une assignation telle que *compte_martin. montant := compte_martin . montant - 10000* (et celui qui fait l'assignation empoche la différence!); cette variable ne peut être changée que de manière indirecte au moyen de l'application, au compte considéré, des procédures exportées de la classe. Cette démarche garantit que l'objet sera manipulé de manière consistante; en particulier, dans le cas présent, une mutation d'un compte n'est possible que si celui qui la fait connait le numéro du compte (de plus, un retrait n'est possible que si ce dernier n'épuise pas le compte). Ainsi, partant de la déclaration précédente de *compte_dupont*, on constate que le retrait *compte_dupont . retirer (1_000_000, 102_658_037)* est impossible à cause d'un crédit insuffisant; par contre, le retrait *compte_dupont. retirer (462, 102 _ 658_037)* a lieu: il laisse le solde de *9_538 Fr.* dans le compte.

Dans ce programme, on remarque la déclaration initiale du type *naturel*; ce type dénote le sous-ensemble de valeurs non négatives du type entier *integer*. Chaque fois qu'une valeur entière est assignée à une variable de type *naturel* (ou qu'un identificateur de valeur de type *naturel* est défini au moyen d'une telle valeur), il est vérifié que cette valeur est non négative; dans le cas contraire,il est imprimé un message et la valeur concernée est remplacée par zéro.

D'une manière générale, un sous-type est défini au moyen d'une déclaration de la forme:

identification_de_type **subrange** identificateur
*(*expression_logique
default suite_d_énoncés **take** expression*)*

L'identificateur qui suit le symbole subrange dénote le sous-type; cependant, à l'intérieur du parenthésage qui clôt sa déclaration, ce même identificateur dénote la valeur présumée du sous-type. Cette valeur est acceptée si l'expression logique qui débute le parenthésage est vraie. La clause par défaut subséquente est optionnelle; en son absence, l'exécution du programme sera interrompue lorsqu'une valeur d'un sous-type ne satisfait pas la contrainte correspondante.

Ce programme *comptes* fait apparaître plusieurs instructions conditionnelles. En général, une instruction conditionnelle a la forme:

suite_de_variantes
variante_par_défaut
done

Il existe plusieurs formes de variantes; dans le programme *comptes*, on a utilisé les deux formes suivantes.

if expression_logique alternative

unless expression_logique alternative

Une alternative a la forme:

then suite_d_énoncés

La variante par défaut a la forme suivante:

default suite_d_énoncés

Comme il apparaît dans l'énoncé conditionnel de la procédure *action*, une instruction conditionnelle peut comporter plusieurs variantes; dans ce cas, ses variantes individuelles sont séparées au moyen du symbole **else**.
Il est important de distinguer les rôles distincts des symboles **else** et **default**.

Le symbole **else** sépare les variantes successives d'un énoncé conditionnel; le symbole **default** introduit la variante par défaut. En fait, **default** pourrait être considéré comme une abréviation de **else if true then**.
Une variante, quelle que soit sa forme, peut être satisfaite ou non satisfaite. L'exécution d'une instruction conditionnelle implique l'élaboration successive de chacune de ses variantes jusqu'à ce que l'on en trouve une qui soit satisfaite. L'alternative incluse dans cette variante est alors élaborée; ceci achève l'exécution de l'ensemble de l'instruction conditionnelle.
On a les règles suivantes:

- Une variante débutant par le mot **if** est satisfaite ssi l'expression logique subséquente est vraie.
- Une variante débutant par le mot **unless** est satisfaite ssi l'expression logique subséquente est fausse.
- La variante par défaut, qui est obtionnelle, est toujours satisfaite.

Remarque:

Dans la procédure *action*, on constate qu'on utilise l'expression *num ~= numéro_compte* pour comparer si deux valeurs *num* et *numéro_compte* du même type sont différentes l'une de l'autre (il aurait aussi été possible d'écrire *num /= numéro_compte*); il faut éviter d'utiliser pour cela l'opérateur <> dont la signification est différente en Newton.

Un type objet peut être récursif; ceci est utile lorsqu'on doit définir des structures de données récursives telles que des listes ou des arbres. Spécifiquement, l'identificateur sur lequel porte une déclaration d'objet est connu dans la partie formelle de cette déclaration. De-même, l'identificateur sur lequel porte une déclaration de classe est connu dans les parties formelles déclarative et exécutable de cette dernière.

Exemple:

```
object liste
    (real value val; liste value suiv);
liste value s =
    liste (3.5, liste (7.9, liste (1.7, nil )))
```

La structure de données issue de *s* est représentée dans la figure 5; on remarque que l'on a construit une liste de trois objets. On a en particulier *s . val = 3.5*, *s .suiv. val = 7.9* et *s . suiv. suiv. val = 1.7*. De plus, *s . suiv. suiv. suiv.* = **nil**; or **nil** est l'objet vide. Cet objet vide appartient conceptuellement à tous les types objets; il ne contient aucune information: il s'ensuit que l'expression s . suiv . suiv . suiv . val n'est pas définie.

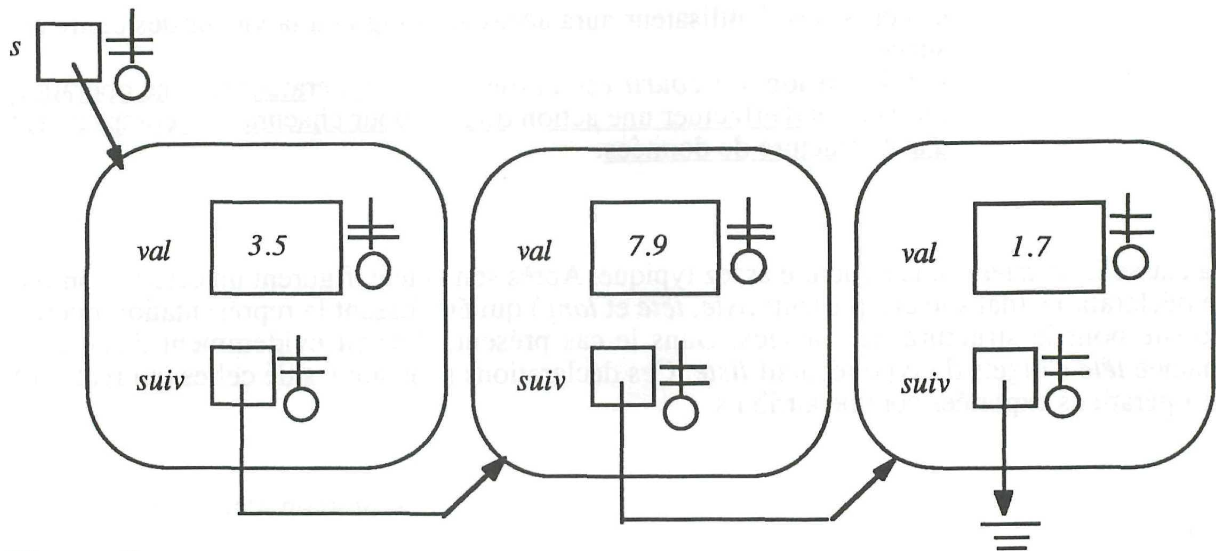


Figure 5

Vu que *s* a été déclaré comme une valeur et que les composantes *val* et *suiv* des objets du type *liste* sont, elles aussi, des valeurs, la structure de données issue de *s* est immuable: elle ne peut être changée ni dans son ensemble (ce qui aurait été possible si *s* était une variable), ni composante par composante (ce qui aurait été possible si *val* et (ou *suiv*) avaient été spécifiés comme variables).

La notion de classe se prête tout aussi bien à la représentation des structures de données classiques qu' à la modélisation de systèmes concrets. On va donner un programme *listes* dans lequel il sera construit des listes de valeurs réelles triées dans l'ordre croissant; ces listes seront implantées au moyen d'une classe *liste_triée*. Cette classe comportera les attributs suivants:

nom — Une chaîne désignant le nom donné à la liste lors de sa création.

longueur — Le nombre de composantes de la liste; initialement, cette valeur est nulle. Par **définition**, *longueur* est un **interrogateur**: un interrogateur est une opération qui permet de consulter l'état d'une structure de données (d'un objet) sans la modifier.

mettre — Insérer un nouvel élément, dont la valeur est fournie en paramètre, dans la liste.
Par **définition**, *mettre* est un **constructeur**: un constructeur est une opération qui permet de construire une structure de données par l'adjonction de nouvelles composantes.

premier — La valeur du premier élément de la liste
Par **définition**, *premier* est un **sélecteur**: un sélecteur est une opération qui permet de consulter la valeur d'un élément spécifique d'une structure de données.

enlever — Eliminer de la liste son premier élément et stocker sa valeur dans une variable fournie en paramètre.
Par **définition**, *enlever* est un **destructeur**: un destructeur est une opération qui permet de détruire une structure de données par élimination de ses composantes.

marier — Réunir en une seule liste la liste considérée et une autre liste fornie comme paramètre.

parcourir — Parcourir la liste élément par élément en faisant une action spécifiée par l'utilisateur sur les composantes successives: au moyen de paramètres

effectifs liés, l'utilisateur aura accès au rang et à la valeur des éléments successifs.
Par **définition,** *parcourir* est un **itérateur**: un itérateur est une opération qui permet d'effectuer une action donnée pour chacune des compsaantes d'une structure de données.

La classe *liste_triée* a une structure assez typique. Après son entête, figurent un certain nombre de déclarations (dans le cas présent: *liste*, *tête* et *long*) qui établissent la représentation interne choisie pour la structure de données. Dans le cas présent, il s'agit évidemment d'une liste chaînée *tête* d'objets du type récursif *liste*. Ces déclarations sont suivies de celles qui réalisent les opérations exportées comme attributs.

```
listes                                                      Vax Newton Compiler 0.2
Page    1
                              Source listing

/* /*OLDSOURCE=USER2:[RAPIN]LISTES.NEW*/ */
PROGRAM listes DECLARE
  CLASS liste_triee
    VALUE moi
    ATTRIBUTE nom,longueur,mettre,premier,enlever,marier,parcourir
    (string VALUE nom)
  (*Un objet de type  liste_triee  est une liste triee, initialement
    vide de valeurs reelles.
  *)
  DECLARE(*liste_triee*)
    (***Representation interne***)
    OBJECT liste(real VALUE val; liste VARIABLE suiv)
                VARIABLE tete:=NIL;
    integer VARIABLE long VALUE longueur:=0;
    (*Le nombre d'elements de la liste*)

    (***Implantation des operations realisables sur la liste***)
    liste_triee FUNCTION mettre
      (real VALUE x)
    (*Insere dans la liste un nouvel element de valeur  x ;
      le resultat est la liste concernee   moi *)
    DECLARE liste REFERENCE ins->tete DO
      CYCLE cherche_place REPEAT
        CONNECT ins THEN
          IF
            x<=val
          EXIT cherche_place DONE;

          ins->suiv
        DEFAULT(*ins=NIL*)
          EXIT cherche_place
        DONE
      REPETITION;
      ins:=liste(x,ins);
      long:=SUCC long
    TAKE moi DONE(*mettre*);

    real EXPRESSION premier=
    (*La valeur du premier element de la liste; en cas de
      liste vide retourne la valeur  INFINITY .
    *)
```

```
      CONNECT tete THEN
        val
      DEFAULT INFINITY DONE;

    liste_triee FUNCTION enlever
      (real REFERENCE xx)
    (*Retire de la liste concernee son premier element et en
      stocke la valeur dans la variable reperee par  xx ; si
      la liste etait initialement vide, stocke  INFINITY
      dans cette derniere sans changer la liste. Resulte dans
      la liste concernee  moi .
    *)
    DO(*enlever*)
      CONNECT tete THEN
        xx:=val; tete:=suiv; long:=PRED long
      DEFAULT xx:=INFINITY DONE
    TAKE moi DONE(*enlever*);

    liste_triee FUNCTION marier
      (liste_triee VALUE toi)
    (*Resulte dans une nouvelle liste contenant la reunion des
      elements de la liste concernee  moi  et de ceux de la liste
      toi ; cette liste aura pour nom la concatenation des noms
      des deux listes jointes par le symbole  "&" . Cette opera-
      tion aura pour effet de vider les listes operandes; elle
      n'a aucun effet si la liste  toi  est la meme liste que
      moi  ou si  toi  est  NIL: le resultat est alors la liste
      concernee  moi .
    *)
    DO(*marier*)TAKE
      IF toi=moi\/toi=NIL THEN
        moi
      DEFAULT(*toi~=moi/\toi~=NIL*)TAKE
        DECLARE
          liste REFERENCE joint->tete;
          liste_triee VALUE union=liste_triee(nom+_"&"_+toi.nom)
        DO
          (*Dans un premier temps, contruit en  tete  la liste
            fusionnee
          *)
          UNTIL toi.tete=NIL REPEAT
            IF TAKE
              CONNECT joint THEN
                toi.tete.val<val
              DEFAULT TRUE DONE
            THEN
              (*Le premier element de la liste fusionnee vient de
                toi.tete
              *)
              joint:=:toi.tete
            DONE;
            (*Le premier element de la fusion est maintenant issu
              de la variable reperee par  joint .
            *)
            joint->joint.suiv
          REPETITION;
          (*La fusion est faite; arrange les resultats*)
          liste[NIL]=:tete=:union.tete;
          (0=:long)+(0=:toi.long)=:union.long
        TAKE
          union
        DONE(*DECLARE joint, union*)
      DONE(*toi~=moi/\toi~=NIL*)
    DONE(*marier*);
```

```

    liste_triee FUNCTION parcourir
      (integer VARIABLE r; real VARIABLE x; PROCEDURE acte)
     (*Parcourt la liste concernee element par element. Pour
       chaque element, stocke son rang dans la variable  r , sa
       valeur dans la variable  x ; effectue ensuite l'enonce
       acte . Le resultat est la liste concernee moi .
     *)
    DECLARE liste VARIABLE curseur:=tete DO
      r:=0;
      WITHIN curseur REPEAT
        r:=SUCC r; x:=val; acte; curseur:=suiv
      REPETITION
    TAKE moi DONE
  DO(*liste_triee*)DONE VALUE
    petite_liste=liste_triee("Mini").mettre(7.4)
                                     .mettre(3.1)
                                     .mettre(5.6),
    romeo=liste_triee("Romeo"), juliette=liste_triee("Juliette");
  /* /*EJECT*/ */
  PROCEDURE construire_liste
    (liste_triee VALUE lt; integer VALUE quantite)
   (*Insere, dans la liste  lt ,  quantite  nouveaux elements
     aleatoires.
   *)
  DECLARE real VARIABLE r DO
    CONNECT lt THEN
      print(page,"*****Construction liste(",nom,")*****");
      FOR integer VALUE k FROM 1 BY 1 TO quantite REPEAT
        IF k\5=1 THEN line DEFAULT print(_) DONE;
        edit((r:=random),13,10); mettre(r)
      REPETITION;
      print(line,"<<<FIN>>>")
    DEFAULT print(line,"###Construction liste inexistante###") DONE
  DONE(*construire_liste*);

  PROCEDURE detruire_liste(liste_triee VALUE lt)
  (*Detruit, element par element, la liste donnee  lt ; la
    valeur de chaque element est imprimee au moment ou il est
    elimine de la liste.
  *)
  DECLARE real VARIABLE elt DO
    CONNECT lt THEN
      print(page,"*****Destruction liste(",nom,")*****",line);
      UNTIL longueur=0 REPEAT
        IF longueur\5=0 THEN line DEFAULT print(_) DONE;
        enlever(elt); edit(elt,13,10)
      REPETITION;
      print(line,"<<<FIN>>>")
    DEFAULT print(line,"###Destruction liste inexistante##") DONE
  DONE(*detruire_liste*);

  PROCEDURE imprimer_liste(liste_triee VALUE lt)DO
  (*Imprime l'etat et les elements de la liste donnee  lt *)
    CONNECT lt THEN
      print(page,"*****Liste triee(",nom,")*****",line);
      print(line,"Nombre d'elements:"_,longueur);
      IF longueur>0 THEN
        print(line,"Premier element:"_,edit(premier,13,10));
        print(line,"***Liste des elements***");
        parcourir(LAMBDA VALUE rang,LAMBDA VALUE valeur,
                  (IF rang\5=1 THEN line DEFAULT print(_) DONE;
                   edit(valeur,13,10)))
      DEFAULT print(line,"---Liste vide---") DONE
```

```
    DEFAULT print(line,"###Impression liste inexistante###") DONE;
    print(line,"<<<FIN>>>")
  DONE(*imprimer_liste*)
  /* /*EJECT*/ */
DO(*listes*)
  construire_liste(romeo,40); construire_liste(juliette,60);
  imprimer_liste(petite_liste);
  imprimer_liste(romeo); imprimer_liste(juliette);
  imprimer_liste(romeo.marier(juliette));
  imprimer_liste(romeo); imprimer_liste(juliette);
  detruire_liste(petite_liste); imprimer_liste(petite_liste);
  print(line,"*****Fin de l'application*****")
DONE(*listes*)

     ****  No messages were issued  ****
```

La figure 6, donnée à la suite des résultats, montre l'état initial de la liste *petite_liste*. On y remarque l'objet engendré par le générateur *liste_triée ("Mini")*: cet objet sert d'entête (ou descripteur) pour l'ensemble de la structure de données. Depuis la variable *tête* de ce descripteur est appondu la liste chaînée des trois éléments du type *liste* introduits à la suite des applications successives *mettre (7.4), mettre (3.1)* et *mettre (5.6)*; ces éléments ont été ordonnés dans l'ordre croissant de leurs composantes *val*. Cette liste n'est, bien entendu, pas directement visible depuis l'extérieur de la classe; elle ne peut être manipulée que par l'intermédiaire des fonctions attributs.

Résultats:

```
*****Construction liste(Romeo)*****
  .6822132664   .8954766199   .6585332471   .5743918847   .5808839519
  .1713170376   .2884567699   .3815839513   .7894531750   .1893110145
  .7055221387   .4500998107   .2834536185   .5627291590   .9940880264
  .2266753487   .0503597815   .9080390208   .6995626790   .9137714516
  .2792520897   .1804553617   .4595076789   .8338560528   .2327648553
  .9546497777   .6387697614   .1169383188   .3250941093   .8670663210
  .4935586209   .6375926567   .0878087549   .2304914224   .6913094543
  .2323697246   .8618420246   .1727308967   .0317200416   .3821486769
<<<FIN>>>


*****Construction liste(Juliette)*****
  .4700551483   .9455096843   .6005856832   .4463034797   .7914942946
  .1050788187   .7604833872   .2949178567   .3037577784   .7844995262
  .6178771809   .8296746614   .1079702336   .9112237146   .7427099949
  .2826084818   .3421278026   .9889511879   .5473246681   .1823816240
  .3649495910   .4444501095   .4150422134   .7050984444   .3567143588
  .6206312330   .5541619924   .2807499796   .8582589663   .7475568610
  .1662768238   .9460602615   .9900418354   .1975602622   .6522765360
  .0977562401   .4767201864   .8639344918   .7423515601   .6431775599
  .1537206342   .7925561542   .2927067127   .5052298213   .1663126921
  .1478423199   .8980292633   .3633930592   .7694178735   .4323917975
  .4237571323   .9053413801   .2796977342   .2602511972   .5583388533
  .3388858515   .9922471635   .3633646364   .5842718652   .7360112816
<<<FIN>>>


*****Liste triee(Mini)*****

Nombre d'elements:          3
Premier element:  3.1000000000
***Liste des elements***
 3.1000000000  5.6000000000  7.4000000000
<<<FIN>>>
```

```
*****Liste triee(Romeo)*****

Nombre d'elements:             40
Premier element:    .0317200416
***Liste des elements***
  .0317200416  .0503597815  .0878087549  .1169383188  .1713170376
  .1727308967  .1804553617  .1893110145  .2266753487  .2304914224
  .2323697246  .2327648553  .2792520897  .2834536185  .2884567699
  .3250941093  .3815839513  .3821486769  .4500998107  .4595076789
  .4935586209  .5627291590  .5743918847  .5808839519  .6375926567
  .6387697614  .6585332471  .6822132664  .6913094543  .6995626790
  .7055221387  .7894531750  .8338560528  .8618420246  .8670663210
  .8954766199  .9080390208  .9137714516  .9546497777  .9940880264
<<<FIN>>>
*****Liste triee(Juliette)*****

Nombre d'elements:             60
Premier element:    .0977562401
***Liste des elements***
  .0977562401  .1050788187  .1079702336  .1478423199  .1537206342
  .1662768238  .1663126921  .1823816240  .1975602622  .2602511972
  .2796977342  .2807499796  .2826084818  .2927067127  .2949178567
  .3037577784  .3388858515  .3421278026  .3567143588  .3633646364
  .3633930592  .3649495910  .4150422134  .4237571323  .4323917975
  .4444501095  .4463034797  .4700551483  .4767201864  .5052298213
  .5473246681  .5541619924  .5583388533  .5842718652  .6005856832
  .6178771809  .6206312330  .6431775599  .6522765360  .7050984444
  .7360112816  .7423515601  .7427099949  .7475568610  .7604833872
  .7694178735  .7844995262  .7914942946  .7925561542  .8296746614
  .8582589663  .8639344918  .8980292633  .9053413801  .9112237146
  .9455096843  .9460602615  .9889511879  .9900418354  .9922471635
<<<FIN>>>


*****Liste triee(Romeo & Juliette)*****

Nombre d'elements:            100
Premier element:    .0317200416
***Liste des elements***
  .0317200416  .0503597815  .0878087549  .0977562401  .1050788187
  .1079702336  .1169383188  .1478423199  .1537206342  .1662768238
  .1663126921  .1713170376  .1727308967  .1804553617  .1823816240
  .1893110145  .1975602622  .2266753487  .2304914224  .2323697246
  .2327648553  .2602511972  .2792520897  .2796977342  .2807499796
  .2826084818  .2834536185  .2884567699  .2927067127  .2949178567
  .3037577784  .3250941093  .3388858515  .3421278026  .3567143588
  .3633646364  .3633930592  .3649495910  .3815839513  .3821486769
  .4150422134  .4237571323  .4323917975  .4444501095  .4463034797
  .4500998107  .4595076789  .4700551483  .4767201864  .4935586209
  .5052298213  .5473246681  .5541619924  .5583388533  .5627291590
  .5743918847  .5808839519  .5842718652  .6005856832  .6178771809
  .6206312330  .6375926567  .6387697614  .6431775599  .6522765360
  .6585332471  .6822132664  .6913094543  .6995626790  .7050984444
  .7055221387  .7360112816  .7423515601  .7427099949  .7475568610
  .7604833872  .7694178735  .7844995262  .7894531750  .7914942946
  .7925561542  .8296746614  .8338560528  .8582589663  .8618420246
  .8639344918  .8670663210  .8954766199  .8980292633  .9053413801
  .9080390208  .9112237146  .9137714516  .9455096843  .9460602615
  .9546497777  .9889511879  .9900418354  .9922471635  .9940880264
<<<FIN>>>
```

```
*****Liste triee(Romeo)*****

Nombre d'elements:            0
---Liste vide---
<<<FIN>>>


*****Liste triee(Juliette)*****

Nombre d'elements:            0
---Liste vide---
<<<FIN>>>

*****Destruction liste(Mini)*****
  3.1000000000  5.6000000000  7.4000000000
<<<FIN>>>
*****Liste triee(Mini)*****

Nombre d'elements:            0
---Liste vide---
<<<FIN>>>
*****Fin de l'application*****
```

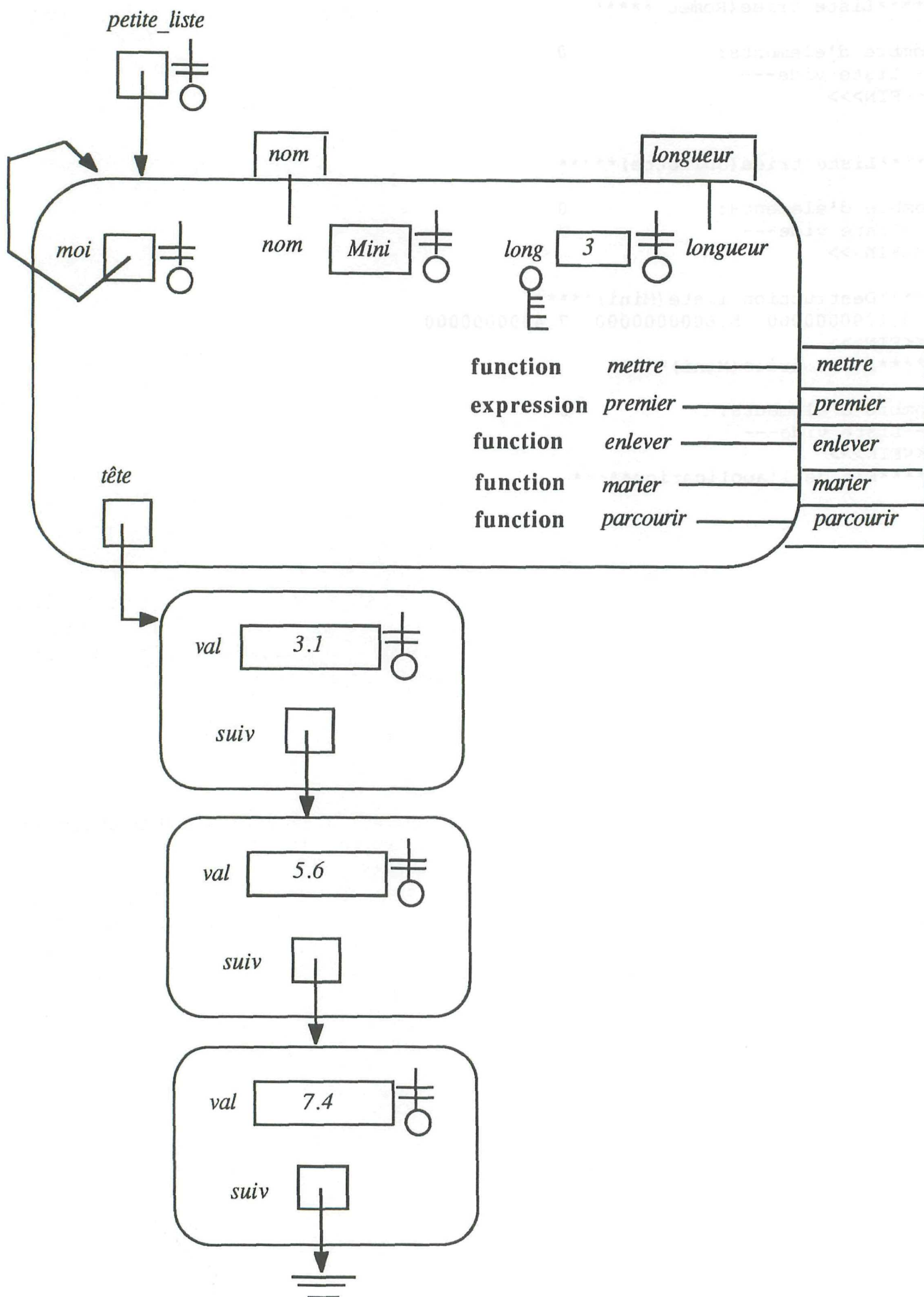

petite_liste
nom
longueur
moi
nom
Mini
long
3
longueur
function mettre
expression premier
function enlever
function marier
function parcourir
mettre
premier
enlever
marier
parcourir
tête
val
3.1
suiv
val
5.6
suiv
val
7.4
suiv

Figure 6

Les objets de la classe *liste_triée* sont auto-identifiés. On remarque, en effet, la clause, **value** *moi* immédiatement après l'identificateur de type *liste_triée*; cette clause signifie que l'identificateur *moi* est connu dans toute la déclaration de classe et qu'il y désigne l'objet spécifique dénoté par l'élaboration considérée de cette dernière. D'une manière générale, toute déclaration de classe ou d'objet peut comporter une partie valeur constituée du symbole **value** suivi d'un identificateur; si présente, cette partie valeur suit immédiatement l'identificateur du type objet: les objets du type correspondants sont alors auto-identifiés au moyen de l'identificateur introduit par cette clause. On constate que le constructeur *mettre*, le destructeur *enlever* et l'itérateur *parcourir* ont été programmés sous la forme de fonctions dont le résultat est l'objet auto-identifié *moi* sur lequel porte l'opération; ceci permet de réaliser, plusieurs opérations successives sur le même objet sans avoir à répéter ce dernier: il en a été fait usage pour initialiser l'objet *petite_liste* au point de sa déclaration au moyen d'une liste de trois éléments.

On remarque qu'une partie valeur peut aussi être incluse dans une déclaration de variable. On considère la déclaration suivante:

integer **variable** *long* **value** *longueur* := 0

Cette déclaration signifie que *long* est une variable entière initialisée à zéro; l'identificateur *longueur* est la valeur courante de cette variable. En spécifiant *longueur*, mais non *long*, comme attribut de la classe *liste_triée* il a été fait en sorte que les utilisateurs de cette classe puissent consulter la valeur de la variable *long* mais non modifier cette dernière.

A l'intérieur de la fonction *mettre*, on remarque l'usage d'une boucle généralisée **cycle**. Les énoncés **while** et **until** permettent facilement de programmer des boucles à point de sortie unique au début, au milieu ou à la fin de la boucle. Pour des boucles plus générales, à points de sortie multiples, on dispose de l'énoncé **cycle**; celui-ci a la forme suivante:

cycle identificateur **repeat**
  suite_d_énoncés
repetition

Le groupe **repeat ... repetition** peut être remplacé par **do ... done**; dans ce dernier cas, les retours en arrière au début de la boucle seront programmés explicitement.

L'identificateur placé après le symbole **cycle** est connu à l'intérieur de la boucle. Il est utilisé, en conjonction avec des formes particulières d'énoncés conditionnels, pour faire sortir de la boucle (clauses de la forme **exit** identificateur_de_cycle) ou répéter cette dernière depuis son début (clauses de la forme **repeat** identificateur_de_cycle). En plus de la forme donnée plus haut, une alternative peut prendre l'une des formes suivantes:

exit identificateur_de_cycle

repeat idnetificateur_de_cycle

then
  suite_d_énoncés
exit identificateur_de_cycle

then
  suite_d_énoncés
repeat identificateur_de_cycle

De-même, une variante par défaut peut prendre les formes suivantes:

default
suite_d_énoncés
exit identificateur_de_cycle
default
suite_d_énoncés
repeat identificateur_de_cycle

On a fait usage, dans la classe *liste_triée* d'énoncés conditionnels **connect** et d'un énoncé répétitif **within**. Une variante peut avoir la forme suivante en plus de celles déjà données:

connect expression_d_object alternative

L'expression d'objet après le symbole **connect** doit livrer un objet d'un type que le compilateur peut déterminer; une telle variante est satisfaite si l'objet résultant de l'évaluation de cette expression est différent de **nil**. Pendant l'exécution de l'alternative subséquente, cet objet est connecté: ceci signifie que ses attributs y sont utilisables sans qu'il soit nécessaire de les qualifier explicitement.Ainsi, dans la procédure *mettre*, l'énoncé conditionnel **if** *x* <= *val* **exit** *cherche_place* **done** signifie **if** *x* <= *ins. val* **exit** *cherche_place* **done** puisque cet énoncé fait partie de l'alternative de la variante introduite par **connect** *ins.*

Un énoncé répétitif **within** est une boucle de la forme suivante:

within expression_d_objet **repeat**
suite_d_énoncés
repetition

De nouveau, l'expression d'objet entre **within** et **repeat** doit être d'un type déterminable à la compilation. La boucle est exécutée tant que cet objet est différent de l'objet vide **nil**; l'objet concerné est connecté pendant l'élaboration de la suite d'énoncés placée entre **repeat** et **repetition**.
On peut constater l'usage de références dans l'implantation de fonctions *mettre* et *marier*; ceci peut se justifier par un processus d'élimination de récursion que l'on va illustrer dans le cas de la fonction *mettre*. En-effet, il est tentant de programmer cette fonction sous une forme récursive calquée sur la structure récursive des listes chaînées sur laquelle elle opère. On pourrait, par exemple, la programmer comme suit:

```
liste_triée function mettre
   (real value x)
declare
   procedure ins_x
      (liste reference ins)
   do
      connect ins then
         if x <= val then
            ins := liste (x, ins)
         default (* x > val *)
            ins_x (suiv)
         done
      default
         ins := liste (x, ins)
      done
   done (* ins_x *)
do
   ins_x (tête);
   long := succ long
take moi done
```

La récursion a été incorporée dans la procédure interne *ins_x*; or cette procédure satisfait au théorème de dérécursification suivant:

Théorème:

Une procédure récursive peut être exprimée de manière purement itérative, sans faire usage de piles, si tous ses appels récursifs interviennent tout à la fin de l'exécution de la procédure (en anglais: tail end recursion).

L'idée consiste à encapsuler le corps de la procédure (y compris sa partie déclarative) dans une boucle. Lorsque l'on sort de la procédure sans passer par une application récursive de cette dernière, on fait sortir de la boucle. Lorsque l'on sort de la procédure à la suite d'un appel récursif, on remplace ce dernier par une suite d'énoncés qui réassocie les paramètres formels de la procédure avec les nouveaux paramètres effectifs puis fait recommencer une nouvelle itération de la boucle.

Il faut faire attention à la manière dont on réassocie les nouveaux paramètres effectifs aux paramètres formels. En cas de passage par valeur, l'association est réalisée par une assignation de la nouvelle valeur du paramètre formel à ce dernier; dans le cas d'un passage par valeur constante, il est nécessaire de remplacer au préalable ce dernier par un passage par valeur (spécifier le paramètre formel comme **variable** et non comme **value**). Dans le cas d'un passage par référence, l'association est réalisée par une dénotation: celle-ci fait repérer la variable communiquée en paramètre effectif par le paramètre formel. C'est ce dernier cas qui intervient si on veut éliminer la récursion de la procédure *ins_x*; appliquant à la lettre le procédé qui vient d'être décrit, cette procédure peut être mise sous la forme itérative suivante:

```
procedure ins_x
   (liste reference ins)
do
   cycle parcours do
      connect ins then
         if x <= val then
            ins := liste (x, ins)
         default (* x > val *)
            ins -> suiv repeat parcours
         done
      default
         ins := liste (x, ins)
      done
   done (* parcours *)
done (* ins_x *)
```

Il n'est ensuite pas très difficile d'éliminer cette procédure en plaçant les actions correspondantes directement dans la partie exécutable de *mettre*, de sortir du cycle l'action terminale *ins := liste (x, ins)* et de remplacer la boucle **cycle** *parcours* **do** par une boucle **cycle** *cherche_place* **repeat** pour arriver à la formulation proposée. Cette formulation itérative sera plus rapide et, surtout, consommera moins de mémoires que la formulation récursive initiale.

Plus loin, on remarque que le sélecteur *premier* a été défini au moyen d'une déclaration d'expression de la forme:

indication_de_type **expression** identificateur = expression

Ceci peut être considéré comme une abréviation d'une déclaration de fonction sans paramètre:

indication_de_type **function** identificateur
   do take expression **done**

Dans la fonction *marier*, on constate l'usage de l'assignation inverse et de la permutation de variables. On y remarque notamment l'énoncé *liste* [**nil**] =: *tête*; la formulation **nil** =: *tête* n'aurait pas été correcte. En-effet, dans une assignation inverse, la variable à droite du symbole =: doit être du même type que l'expression à gauche de ce symbole; or **nil** n'appartient pas spécifiquement au type *liste* : il appartient au type prédéfini *pointer* qui inclut tous les types objets. La variable *tête* est du type objet *liste* et non du type *pointer*; comme il n'y a pas de conversion implicite de type au niveau de variable, seulement parfois au niveau des valeurs, l'assignation inverse **nil** =: *tête* est incorrecte. L'expression *liste* [**nil**] est un forceur; d'une manière générale, un forceur a l'une des formes:

type [expression]

type [séquence_d_énoncés **take** expression]

L'expression entre crochets doit produire une valeur d'un type susceptible d'être converti implicitement à une valeur du type désigné avant le crochet-gauche; le résultat du forçage est la valeur ainsi convertie. Dans le cas présent, le forçage *liste* [**nil**] dénote l'objet vide du type *liste*.

Comme autre particularité, on remarque la manière dont le type *liste* et la variable *tête* ont été déclarés:

object *liste* (*real* **value** *val*; *liste* **variable** *suiv*)
variable *tête* := **nil**

Cette déclaration compactée est équivalente à la paire de déclarations suivante:

object *liste* (*real* **value** *val*; *liste* **variable** *suiv*);
liste **variable** *tête* := **nil**

D'une manière générale, la déclaration d'un nouveau type d'information est toujours utilisable comme indication de type. On remarque que l'on a aussi utilisé cette propriété, à la suite de la classe *liste_triée*, pour déclarer les valeurs *petite_liste*, *romeo* et *juliette*.

La procédure *marier* fait intervenir un bloc local; c'est-à-dire un énoncé **declare**; celui-ci permet d'introduire un nouveau niveau de déclarations: il a la forme:

```
declare
   suite_d_déclarations
do
   suite_d_énoncés
done
```

Dans le cas présent, cet énoncé doit produire un résultat d'un type connu à priori, ce qui explique l'insertion d'une clause **take** expression entre la suite d'énoncés et le **done**.

Pour la compréhension de ce programme, on peut encore signaler que l'opérateur arithmétique \ dénote le reste de la division entière de ses deux opérandes. Le catalogue des opérations disponibles sur les valeurs arithmétiques entières et réelles sera donné au chapitre suivant.

Chapitre 3

Arithmétique

Le type prédéfini *integer* inclut toutes les valeurs entières comprises entre deux bornes *integer* **min** et integer **max.** Sur le VAX, on a *integer* **max** = *2_147_483_647* tandis que *integer* **min** est égal à l'opposé de cette valeur.

Plusieurs opérateurs sont disponibles sur les valeurs entières; comme pour les valeurs logiques, on a différents groupes d'opérateurs de priorités différentes. Par ordre de priorité dcroissant, on a les opérations suivantes dans lesquelles on suppose que *j* et *k* dénotent des valeurs entières.

Groupe 1. Création de caractères alphanumériques

letter *k*
- si $1 <= k \wedge k <= 26$:
  la *k_e* lettre majuscule
  Exemple: **letter** *5* = *"E"*
- si $27 <= k \wedge k <= 52$:
  la *(k - 26)_e* lettre minuscule
  Exemple: **letter** *30* = *"d"*
- non défini pour $k <= 0 \vee k > 52$

digit *k*
- si $0 < k \wedge k <= 9$:
  le chiffre de rang *k*
  Exemple: **digit** *6* = *"6"*
- si $10 <= k \wedge k <= 61$:
  même chose que **letter** *(k - 9)*
  Exemple: **digit** *12* =*" C"*
- non défini si $k < 0 \vee k > 61$

Groupe 2. Prise de signe

sign *k*
- si *k > 0,* **sign** *k = 1*
- si *k = 0,* **sign** *k = 0*
- si *k < 0,* **sign** *k = -1*

Groupe 3. Opérateurs multiplicatifs

*j * k* produit entier
Exemple: *5 * 7 = 35*

j / k quotient réel
Exemple: *75 / 8 = 9.375*

j % k quotient entier tronqué vers le bas.
Exemples: *75 % 8 = 9*
(- 75) % 8 = - 10
75 % (- 8) = - 10
(- 75) % (- 8) = 9

j \ k reste du quotient entier; ce reste, sauf s'il est nul, a le signe du diviseur.
Exemples: *75 \ 8 = 3*
(- 75) \ 8 = 5
75 \ (- 8) = - 5
(- 75) \ (- 8) = - 3

Groupe 4. Opérateurs additifs monadiques

$+ k$	identité
$- k$	changement de signe
pred *k*	l'entier précédent **Exemple:** **pred** $5 = 4$
succ *k*	l'entier suivant **Exemple:** **succ** $8 = 9$
abs *k*	valeur absolue **Exemples:** **abs** $7 = 7$ **abs** $(- 3) = 3$

Groupe 5. Opérateurs additifs dyadiques

$j + k$	addition entière **Exemple:** $5 + 7 = 12$
$j - k$	soustraction entière **Exemple:** $5 - 7 = - 2$

Groupe 6. Opérateurs d'optimisation

j **min** *k*	minimum entier **Exemple:** 5 **min** 7 = 5
j **max** *k*	maximum entier **Exemple:** 5 **max** 7 = 7
Remarque:	Les opérations **min** et **max** sont disponibles pour tous les types ordonnés (*integer, real, character, string*, types scalaires)

Groupe 7. Interrogateurs

even *k*	vrai ssi *k* est pair **Exemples:** **even** 8 ~ **even** 5
odd *k*	vrai ssi *k* est impair **Exemple:** **odd** 7

Groupe 8. Comparaisons

On dispose de six comparaisons usuelles:

$j = k$	vrai ssi	*j*	est égal à	*k*
$j \sim= k$	vrai ssi	*j*	est différent de	*k*
$j < k$	vrai ssi	*j*	est inférieur à	*k*
$j <= k$	vrai ssi	*j*	ne dépasse pas	*k*
$j > k$	vrai ssi	*j*	est supérieur à	*k*
$k >= k$	vrai ssi	*j*	est au moins égal à	*k*

<u>Une valeur entière peut,</u> de plus , <u>être forcée au type *real*, au type *character* ou à un type scalaire</u>.

Exemples:

real [562] = 562.0

Sachant que le code Ascii de la lettre *"A"* est égal à *65*, c'est-à-dire sachant que **ord** *"A" = 65*, le forçage au type character donne l'opération inverse *character [65] = "A"*.
Soit le type scalaire *couleur* résultant de la déclaration suivante:

scalar *couleur (rouge, orange, jaune, vert, bleu, violet)*

On a par exemple **ord** *vert = 3*; de nouveau, le forçage d'une valeur entière au type couleur réalise l'opération inverse du **ord:** *couleur [3] = vert.*

Le type *real* inclut un sous ensemble discret de valeurs réelles. Ce sous-ensemble inclut:

- la valeur *0.0*
- les valeurs **infinity** et - **infinity**
- un sous-ensenble de nombres dont la valeur absolue est comprise entre deux bornes *real* **min** et *real* **max**; l'écart relatif entre deux valeurs réelles successives représentées exactement est de l'ordre de *epsilon.* Sur le VAX, on a approximativement *real* **min** = *.29 & - 38*, *real* **max** = *1.7 & 38* et *epsilon = 1.4 & - 17.* Les valeurs réelles seront donc incluses, en valeurs absolue, entre 10^{-38} et 10^{+38}; elles sont définies avec environ *16* chiffres décimaux significatifs.

Trois valeurs réelles particulières peuvent être nommées au moyen d'identificateurs prédéfinis:

pi = 3.1415926535897932
e = 2.7182818284590453
gamma = . 57721566490153286

Cette dernière constante est la constante d'Euler définie comme suit:

$$\gamma = \lim_{n \to \infty} \left(\sum_{k=1}^{n} (1/k) - \ln (n) \right)$$

De nouveau, plusieurs opérations sont disponibles sur les valeurs réelles; certaines réflètent d'ailleurs les opérations entières correspondantes. Dans les groupes suivants, *x* et *y* dénoteront des valeurs réelles et *k* une valeur entière.

Groupe 1. Exponentiation

*x * * k* **Exemples:** *2.3 * * 2 = 5.29*
*(- 0.8) * * (- 3) = - 1.953125*

Groupe 2. Opérateurs multiplicatifs dyadiques

sign *x*	Prise de signe	
	Exemples:	**sign** *5.06 = 1*
		sign *(- 3.78) = -1*
floor *x*	Partie entière (le plus grand entier contenu dans *x*)	
	Exemples:	**floor** *6.18 = 6*
		floor *(-6.18) = - 7*
ceil *x*	Le plus petit entier au moins égal à *x*	
	Exemple:	**ceil** *6.18 = 7*
		ceil *(- 6.18) = - 6*
frac *x*	Partie fractionnaire	
	Exemples:	**frac** *6.18 = . 18*
		frac *(- 6.18) = .82*
	Remarque:	On a *x* = **floor** *x* + **frac** *x*

Groupe 3. Opérateurs multiplicatifs dyadiques

*x * y*	produit réal	
x / y	quotient réel	
x % y	quotient entier (la partie entière du quotient réel)	
	Exemples:	*7.85 % 1.23 = 6*
		(- 7.85) % 1.23 = - 7
		7.85 % (- 1.23) = - 7
		(- 7.85) % (- 1.23) = 6
x \ y	reste du quotient entier	
	Exemples:	*7.85 \ 1.23 = .47*
		(- 7.85) \ 1.23 = . 76
		7.85 \ (- 1.23) = - . 76
		(- 7.85) \ (- 1.23) = - . 47

Groupe 4. Opérateurs additifs monadiques

+ *x*	identité
- *x*	changement de signe
abs *x*	valeur absolue

Groupe 5. Opérateurs additifs dyadiques

x + y	addition réelle
x - y	soustraction réelle

Groupe 6. Opérateurs d'optimisation

x **min** *y*	minimum réel
x **max** *y*	maximum réel

Groupe 7. Interrogateur

finite *x*	vrai ssi *x* a une valeur finie	
	Exemple:	**finite** *2.86*
		~ **finite** *(-* **infinity***)*

Groupe 8. Comparaison

$x = y$	
$x \sim= y$	
$x < y$	
$x <= y$	Signification usuelle
$x > y$	
$x >= y$	

D'autres opérations sont exprimables au moyen de fonctions prédéfinies:

sqrt (x)	racine carrée	
	Condition d'emploi:	$x \geq 0$
exp (x)	*e* élevé à la puissance *x*	
ln (x)	logarithme népérien	
	Conditions d'emploi	$x > 0$
	ln (0) peut donner	- **infinity**
sin (x)	sinus (*x* en radians)	
cos (x)	cosinus (*x* en radians)	
arctan (x)	arc tangente (détermination comprise entre *-pi / 2* et *+ pi / 2*)	
random	valeur aléatoire tirée, selon distribution uniforme dans *[0.0, 1.0[*	
normal	valeur aléatoire tirée selon distribution normale de moyenne nulle et d'écart-type égal à *1*.	
poisson	valeur aléatoire tirée selon distribution exponentielle négative de moyenne égale à *1*.	

Remarque:

Les générateurs aléatoires *random*, *normal* et *poisson* fournissent des valeurs pseudo-aléatoires; ils sont implantés au moyen d'un algorithme: cet algorithme prend pour donnée un germe (contenu dans une variable globale invisible) et change la valeur du germe à chaque tirage. Il en résulte que plusieurs passages d'un même programme donnent normalement lieu à la même suite de tirages aléatoires: les résultats sont reproductibles. Cette propriété est utile en phase de mise au point d'un application. Par contre, il peut être nécessaire d'exploiter le même programme plusieurs fois avec des suites de tirages aléatoires différentes: on dispose pour cela de la procédure prédéfinie *randomize* dont l'effet est d'initialiser de manière imprévisible le germe des valeurs aléatoires.

Une valeur réelle peut être forcée au type entier; ce forçage implique une conversion au plus proche entier (il y a donc arrondi).

Exemples:

integer [3.52] = 4
integer [3.15] = 3
integer [- 6.8] = - 7
integer [8.5] = n'est pas défini:
le résultat peut être *8* ou *9*.

Chapitre 4

Rangées

En lieu et place de tableaux, on a recours en Newton à la notion de rangée. Une rangée est un objet contenant un ensemble de variables de même type indicées au moyen d'un ensemble contigu de valeurs entières. Un type rangée sera déclaré de la manière suivante

indication_de_type **row** identificateur

Exemple:

real **row** *vecteur* **row** *matrice*

Dans cet exemple, *vecteur* dénote des rangées à composantes réelles et *matrice* des rangées dont les composantes sont du type *vecteur.*

Les types rangées présentent deux différences essentielles par rapport aux tableaux que l'on trouve, par exemple, en Pascal.

- Les bornes des indices ne font pas partie du type; elles appartiennent à chaque rangée individuelle. Deux rangées du même type auront donc, en général, des indices de bornes différentes.
- Comme pour n'importe quel objet, les rangées sont crées dynamiquement à l'exécution du programme. En particulier, les bornes de leurs indices peuvent dépendre de données qui ne sont connues qu'à l'exécution du programme.

Une rangée est créée au moyen d'un générateur de rangée de la forme:

type_rangée *(*expression_entière **to** expression_entière*)*

La première expression livre la valeur de la borne inférieure des indices et la seconde celle de leur borne supérieure.

Exemple:

real **row** *vecteur* **value**
v1 = vecteur (1 **to** *3),*
v2 = vecteur (0 **to** *99)*

Cette déclaration établit deux rangées du type *vecteur*; la première *v1* comporte trois composantes réelles (les indices correspondants seront compris entre *1* et *3*) tandis que la seconde en comporte *100* avec des indices compris entre *0* et *99.*

Soit *r* une rangée et *k* une valeur entière, on dispose des opérations suivantes:

Interrogateurs:

low *r*	la borne inférieure des indices de *r*
high *r*	la borne supérieure des indices de r
	Exemple: **low** *v1 = 1*
	high *v2 = 99*
	(où l'on présuppose les rangées *v1* et *v2* déclarées plus haut)

Sélecteur:

r [k]	la composante d'indice *k* de la rangée *r*.
	Condition d'emploi:
	low *r* < = *k* ∧ *k* <= **high** *r*
	Le résultat de cet opération d'indiçage est évidemment une variable.

Itérateur:

```
through r
   index k
   value rk
   reference rrk
   := expression
repeat
   suite_d_énoncés
repetition
```

L'idée est d'effectuer la suite d'énoncés incluse entre **repeat** et **repetition** pour chacune des composantes de chacune des composantes de la rangée *r*. Les clauses **index** *k*, **value** *rk*, **reference** *rrk* et := expression sont toutes facultatives. Le cas échéant, l'identificateur *k* désigne la valeur de l'indice de la composante considérée et *rk* la valeur de la composante elle-même. De-même, *rrk* est un repère à la composante considérée. Finalement, la valeur de l'expression après le symbole := est, le cas échéant, stockée dans la composante considérée. L'ensemble de l'itérateur **through** peut être utilisé comme expression; il produit alors pour résultat la rangée concernée. Cette propriété peut être utilisée pour initialiser une rangée au moment de sa déclaration.

Exemple:

```
real row vecteur row matrice value
   m = through
      matrice (1 to 3)
         index j :=
            through
               vecteur (1 to j)
               index k := (j + k) / 2
            repetition
         repetition
```

Cette déclaration établit en *m* une matrice triangulaire d'ordre trois; cette matrice est représentée à la figure 7.

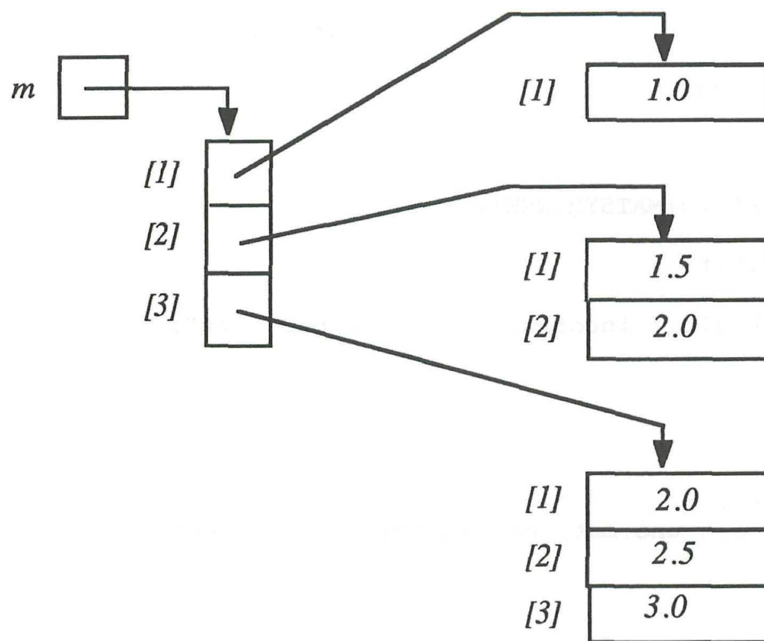


Figure 7

Il ressort de cet exemple que le mot **repeat** peut être omis lorsqu'il n'y a qu'une instruction vide entre **repeat** et **repetition.**

Comparaisons:

Deux rangées du même type peuvent être comparées pour l'égalité ou l'inégalité au moyen des opérateurs = et ~=. Il convient de faire attention au fait qu'il s'agit d'une comparaison d'objets: deux rangées seront considérées comme égales si il s'agit de la même rangée.

Exemple:

```
real row vecteur value
   v1 =  through
             vecteur (1 to 4) index j : = j / 2
         repetition,
   v2 =  through
             vecteur (1 to 4) index j := j / 2
         repetition
```

Après ces déclarations, la relation *v1* = *v2* sera fausse; *v1* et *v2* sont deux rangées différentes, bien qu'elles aient mêmes bornes et que les composantes homologues aient le même contenu.

Dans le programme *matsym*, on montre l'implantation de matrices symétriques au moyen de matrices triangulaires selon le principe illustré, dans le cas d'une matrice d'ordre trois, à la figure 7. L'implantation est décrite dans la classe *mat_sym*; cette classe fait intervenir une fonction d'accès et une clause d'indiçage.

```
matsym                                      Vax Newton Compiler 0.2
Page    1
                    Source listing

/* /*OLDSOURCE=USER2:[RAPIN]MATSYM.NEW*/ */
PROGRAM matsym DECLARE
  integer SUBRANGE positif
    (positif>0 DEFAULT
       print(line,"###Positif incorrect remplace par 1###")
     TAKE 1);

  CLASS mat_sym
    INDEX elt
    ATTRIBUTE ordre
    (positif VALUE ordre)
  (*Un objet  mat_sym  est une matrice symetrique de dimension
    ordre  donnee.
  *)
  DECLARE(*mat_sym*)
    real ROW vecteur ROW matrice VALUE m=
      THROUGH
        matrice(1 TO ordre)
        INDEX j
        :=THROUGH vecteur(1 TO j) REPETITION
      REPETITION;

    real ACCESS elt
      (positif VALUE i,j)
    (*La composante d'indices  i  et  j *)
    DECLARE positif VALUE ii=i MAX j,jj=i MIN j DO
      IF ii>ordre THEN
        print(line,"###Erreur d'indicage###")
      RETURN DONE
    TAKE m[ii][jj] DONE(*elt*)
  DO(*mat_sym*)DONE VALUE
    s1=mat_sym(3),s2=mat_sym(6);

  PROCEDURE init_mat_sym
    (mat_sym VALUE s)
  (*Initialise avec des elements aleatoires la matrice  s *)
  DECLARE real VARIABLE r DO
    print(page,"***Construction d'une matrice symetrique***");
    FOR integer VALUE i FROM 1 TO s.ordre REPEAT
      FOR integer VALUE j FROM 1 TO i REPEAT
        IF j\5=1 THEN line DONE;
        edit((r:=random),12,8);
        s[i,j]:=r
      REPETITION;
      line
    REPETITION;
    print("<<<FIN>>>",line)
  DONE(*init_mat_sym*);

  PROCEDURE impr_mat_sym
    (mat_sym VALUE s)
  (*Imprime les composantes de la matrice  s *)
```

```
matsym                                              Vax Newton Compiler 0.2
Page    2
                         Source listing

   216  DO(*impr_mat_sym*)
   217    print(line,"***Contenu d'une matrice symetrique***");
   224    FOR integer VALUE i FROM 1 TO s.ordre REPEAT
   235      FOR integer VALUE j FROM 1 TO s.ordre REPEAT
   246        IF j\5=1 THEN line DONE; edit(s[i,j],12,8)
   269      REPETITION;
   271      line
   272    REPETITION;
   274    print("<<<FIN>>>",line)
   280  DONE(*impr_mat_sym*)
   281 DO(*matsym*)
   282   randomize(*pour changer un peu*);
   284   init_mat_sym(s1); impr_mat_sym(s1);
   294   init_mat_sym(s2); impr_mat_sym(s2)
   303 DONE(*matsym*)

      ****  No messages were issued  ****
```

Une fonction d'accès est une fonction dont le résultat est une variable (et non une valeur) du type considéré. Il s'ensuit qu'une fonction d'accès est utilisable dans tout contexte où une variable a un sens, y compris à gauche du symbole d'assignation := .

Résultats:

```
***Construction d'une matrice symetrique***
   .04585757

   .59134892    .20874674

   .57891792    .97430574    .35490540
<<<FIN>>>

***Contenu d'une matrice symetrique***
   .04585757    .59134892    .57891792

   .59134892    .20874674    .97430574

   .57891792    .97430574    .35490540
<<<FIN>>>

***Construction d'une matrice symetrique***
   .75831294

   .87803545    .77542439

   .26992825    .63368489    .85229252

   .07299191    .16517401    .09419816    .33666929

   .40684954    .95582782    .37235815    .49268933    .30581511

   .96749717    .24564256    .17663118    .62309698    .71304585
   .08793376
<<<FIN>>>
```

```
***Contenu d'une matrice symetrique***
    .75831294    .87803545    .26992825    .07299191    .40684954
    .96749717

    .87803545    .77542439    .63368489    .16517401    .95582782
    .24564256

    .26992825    .63368489    .85229252    .09419816    .37235815
    .17663118

    .07299191    .16517401    .09419816    .33666929    .49268933
    .62309698

    .40684954    .95582782    .37235815    .49268933    .30581511
    .71304585

    .96749717    .24564256    .17663118    .62309698    .71304585
    .08793376
<<<FIN>>>
```

Exemple:

> *real* **variable** *x, y;*
> *real* **access** *x_or_y*
>   *(integer* **value** *n)*
> **do take**
>   **if** **even** *n* **then** *x* **default** *y* **done**
> **done** *(*x_or_y*)*

L'énoncé *x_or_y (17) := 47.2* aura pour effet de stocker la valeur *47.2* dans la variable *y* retournée comme résultat de l'application *x_or_y (17)* puisque le paramètre effectif *17* est impair.

La forme générale d'une fonction d'accès est semblable à celle d'une fonction; le mot **access** remplace **function** et le résultat doit être livré sous la forme d'une variable du type approprié et non d'une expression arbitraire.

> indication_de_type **access** identificateur
>   paramètres_formels
> **declare**
>   suite_de déclarations
> **do**
>   suite_d_énoncés
> **take**
>   variable
> **done**

Une classe peut comporter une clause d'indiçage placée, le cas échéant, après la partie valeur éventuelle et avant la liste des attributs. Cette clause a la forme:

> **index** identificateur

L'identificateur qui suit le symbole **index** doit être déclaré dans la partie déclarative de la classe; cet identificateur doit obligatoirement désigner une fonction ou une fonction d'accès possédant une liste de paramètres formels non vide.

Lorsqu'une classe possède une clause d'indiçage, les objets correspondants admettent une opération d'indiçage. Cette opération est implantée par une application de la fonction ou fonction d'accès désignée par la clause d'indiçage aux indices donnés dans cette opération.

Ainsi, sans le cas de la classe *mat_sym* du programme *matsym*, si l'on a un objet *s* du type *mat_sym*, cet objet peut être indicé au moyen d'une partie des valeurs entières qui seront considérées comme paramètres effectifs de la fonction d'accès *elt*. Soient donc *i* et *j* deux valeurs entières positives, l'expression *s [i, j]* serait équivalente à *s . elt (i, j)* : cette dernière notation n'est cependant pas acceptable en dehors de la classe *mat_sym* puisque *elt* ne figure pas dans sa liste d'attributs. Vu que *elt* est une fonction d'accès, l'entité *s [i, j]* est évidemment une variable: elle est donc utilisable à gauche du symbole *:=* ce qui apparaît clairement dans la procédure *init_mat_sym*.

A l'intérieur de la fonction d'accès *elt*, on remarque la clause **return** élaborée en cas d'erreur d'indiçage. Dans ce contexte, elle aura pour effet d'interrompre le programme et de retourner le contrôle au système d'exploitation. On verra plus tard que cette clause est liée aux manipulations de coroutines et qu'elle peut avoir un effet plus général.

Une autre manière d'implanter une matrice symétrique, un peu plus économe en mémoire, mais (probablement) un peut plus coûteuse en temps de calcul, consiste à n'utiliser qu'une seule rangée de dimension *ordre * (ordre + 1) % 2*. Soit *ii* et *jj* la paire d'indices avec *ii >= jj*, la composante correspondante sera placée dans l'élément d'indice *jj + (ii - 1) * ii % 2* de la rangée représentative. Cet indice est obtenu facilement en partant de l'indice *ii * (ii + 1) % 2* de l'élément diagonal de rang *ii* et en en retranchant la différence *ii -jj* des deux indices. La version modifiée suivante du programme *matsym* illustre cet autre choix de représentation. Vu le recours à la procédure *randomize*, on obtient à chaque application de l'une ou l'autre version du programme *matsym* des résultats différents, mais de même structure générale. L'exemplaire qui suit la deuxième version du programme *matsym* a été obtenu avec ce dernier programme.

Pour le reste, cette deuxième version du programme *matsym* illustre la facilité avec laquelle on a pu passer d'une des représentations internes à l'autre au moyen de changements très localisés dans la classe *mat_sym*.

```
matsym                                                    Vax Newton Compiler 0.2
Page    1
                    Source listing

    1 /* /*OLDSOURCE=USER2:[RAPIN]MATSYM2.NEW*/ */
    1 PROGRAM matsym DECLARE
    4   integer SUBRANGE positif
    7     (positif>0 DEFAULT
   12       print(line,"###Positif incorrect remplace par 1###")
   18     TAKE 1);
   22
   22   CLASS mat_sym
   24     INDEX elt
   26     ATTRIBUTE ordre
   28     (positif VALUE ordre)
   33   (*Un objet  mat_sym  est une matrice symetrique de dimension
   33     ordre  donnee.
   33   *)
   33   DECLARE(*mat_sym*)
   34     real ROW vecteur VALUE v=vecteur(1 TO ordre*(ordre+1)%2);
   55
   55     real ACCESS elt
   58       (positif VALUE i,j)
   65     (*La composante d'indices  i  et  j *)
   65     DECLARE positif VALUE ii=i MAX j,jj=i MIN j DO
   80       IF ii>ordre THEN
   85         print(line,"###Erreur d'indicage###")
   91       RETURN DONE
   93     TAKE v[jj+(ii-1)*ii%2] DONE(*elt*)
  109   DO(*mat_sym*)DONE VALUE
  112     s1=mat_sym(3),s2=mat_sym(6);
  126
  126   PROCEDURE init_mat_sym
  128     (mat_sym VALUE s)
  133   (*Initialise avec des elements aleatoires la matrice  s *)
  133   DECLARE real VARIABLE r DO
  138     print(page,"***Construction d'une matrice symetrique***");
  145     FOR integer VALUE i FROM 1 TO s.ordre REPEAT
  156       FOR integer VALUE j FROM 1 TO i REPEAT
  165         IF j\5=1 THEN line DONE;
  175         edit((r:=random),12,8);
  188         s[i,j]:=r
  196       REPETITION;
  198       line
  199     REPETITION;
  201     print("<<<FIN>>>",line)
  207   DONE(*init_mat_sym*);
  209
  209   PROCEDURE impr_mat_sym
  211     (mat_sym VALUE s)
  216   (*Imprime les composantes de la matrice  s *)
  216   DO(*impr_mat_sym*)
  217     print(line,"***Contenu d'une matrice symetrique***");
  224     FOR integer VALUE i FROM 1 TO s.ordre REPEAT
  235       FOR integer VALUE j FROM 1 TO s.ordre REPEAT
  246         IF j\5=1 THEN line DONE; edit(s[i,j],12,8)
  269       REPETITION;
  271       line
  272     REPETITION;
  274     print("<<<FIN>>>",line)
  280   DONE(*impr_mat_sym*)
```

```
DO(*matsym*)
  randomize(*pour changer un peu*);
  init_mat_sym(s1); impr_mat_sym(s1);
  init_mat_sym(s2); impr_mat_sym(s2)
DONE(*matsym*)

     ****  No messages were issued  ****
```

Résultats:

```
***Construction d'une matrice symetrique***
   .17667581

   .22073185   .40516994

   .07142667   .75319819   .06239933
<<<FIN>>>

***Contenu d'une matrice symetrique***
   .17667581   .22073185   .07142667

   .22073185   .40516994   .75319819

   .07142667   .75319819   .06239933
<<<FIN>>>


***Construction d'une matrice symetrique***
   .47174674

   .66252439   .42535098

   .87368250   .99705666   .75130851

   .40723708   .21137534   .13641650   .78562003

   .23552337   .16802159   .54900082   .93481362   .94206620

   .04862890   .36024988   .25195418   .10434143   .07685161
   .97914215
<<<FIN>>>

***Contenu d'une matrice symetrique***
   .47174674   .66252439   .87368250   .40723708   .23552337
   .04862890

   .66252439   .42535098   .99705666   .21137534   .16802159
   .36024988

   .87368250   .99705666   .75130851   .13641650   .54900082
   .25195418

   .40723708   .21137534   .13641650   .78562003   .93481362
   .10434143

   .23552337   .16802159   .54900082   .93481362   .94206620
   .07685161

   .04862890   .36024988   .25195418   .10434143   .07685161
   .97914215
<<<FIN>>>
```

Chapitre 5

Objets procéduraux

A l'instar de langages tels que Algol-68 et Modula 2, Newton permet de manipuler procédures et fonctions par l'intermédiaire de types procéduraux: les valeurs correspondantes sont des objets procéduraux. Par l'intermédiaire d'objets procéduraux, il est possible de créer, dynamiquement, à l'exécution du programme, de nouvelles procédures ou fonctions; il est ensuite possible de les faire exécuter en leur associant des paramètres effectifs du type et de la forme appropriés.

La déclaration d'un type procédural peut prendre l'une des trois formes suivantes:

> **actor** identificateur paramètres_formels
>
> indication_de_type **functor** identificateur
>   paramètres_formels
>
> indication_de_type **accessor** identificateur
>   paramètres_formels

Exemples:

> **actor** *truc (integer* **value** *j);*
> **actor** *action;*
> *real* **functor** *fonction (real* **value** *x)*
>   **functor** *fonctionnelle (fonction* **value** *f);*
> *character* **accessor** *charax (integer* **reference** *jj, kk)*

Cette ensemble des déclarations définit cinq types procéduraux:

- Le type acteur *truc*: les objets procéduraux correspondants seront des procédures à un paramètre entier transmis par valeur constante (ou par valeur).
- Le type acteur *action*: les objets procéduraux correspondants seront des procédures sans paramètres.
- le type foncteur *fonction*: les objets procéduraux (ou objets fonctionnels) correspondants seront des fonctions à un paramètre réel transmis par valeur constante (ou par valeur) et à résultat réel.
- Le type foncteur *fonctionnelle*: les objets procéduraux correspondants seront des fonctions à un paramètre du type *fonction* transmis par valeur constante (ou par valeur) et à résultat de ce même type *fonction*.
- Le type accesseur *charax*: les objets procéduraux correspondants seront des fonctions d'accès admettant deux paramètres entiers transmis par référence et dont le résultat est une variable du type *character*.

Il résulte de cette discussion que les types acteurs représentent des procédures pures, les types foncteurs des fonctions et les types accesseurs des fonctions d'accès. Avant de passer à des applications pratiques, on va montrer, par des exemples, comment fabriquer des objets procéduraux. Par cela, on présuppose les types procéduraux *truc*, *action*, *fonction*, *fonctionnelle* et *charax* définis plus haut.

```
procedure imprime_ok do
   print (line, "<<< OK >>>")
done;
action value
   pok = action (imprime_ok),
   pal = body action do print (line, random) done;

procedure imprime_int
   (integer value j)
do print (line, j) done;
truc value
   pint = truc (imprime_int),
   pcar =    body truc (integer value j) do
                 pint [j * * 2]
             done;
real function pki
   (real value t)
do take (1 - t) / (1 + t) done;
fonction value
   rapport = fonction (pki),
   rapcar = body fonction declare
                 real value x2 = x * * 2
             do take (1 - x2) / (1 + x2) done;
fonctionnelle value partie_paire =
   body fonctionnelle (fonction value f) do
   take body fonction (real value x) do
         take (f [x] + f [- x]) / 2 done
   done;
function value pairapport = partie_paire [rapport];
character row chars value texte = chars (1 to 80);
character access tabac
   (integer reference u, v)
do
   u := succ u; v := pred v
take texte [(u + v) % 2] done;
charax value tac = charax (tabac )
```

De cet exemple, il ressort qu'il y a deux manières principales de créer des objets procéduraux.

1 . Au moyen d'un générateur d'objet procédural, une procédure, une fonction ou une fonction d'accès est transformée en un objet d'un type acteur, foncteur ou respectivement accesseur approprié. Les formes respectives sont les suivantes:

type_acteur (identificateur_de_procédure)
type_foncteur (identificateur_de_fonction)
type_accesseur (identificateur_de_fonction_d_accès)

Un tel générateur est correct si les conditions suivantes sont satisfaites:

- La partie formelle de la procédure, fonction ou fonction d'accès est compatible avec celle du type procédural concerné.
- Dans le cas d'un type foncteur ou accesseur, le type du résultat de ce dernier est identique au type de la fonction ou respectivement de la fonction d'accès sur laquelle porte le générateur.
- La partie formelle de la procédure, fonction ou fonction d'accès sur laquelle porte un générateur d'objet procédural est compatible avec celle du type procédural concerné si les conditions suivantes sont satisfaites.

- Le nombre de paramètres formels est le même.

- Les types des paramètres formels homologues sont identiques.

- Les modes de transmission des paramètres formels homologues sont identiques. Par exception, un paramètre formel spécifié comme **value** dans un type procédural peut correspondre à un paramètre formel spécifié comme **variable** dans la partie formelle de la procédure, fonction ou fonction d'accès sur laquelle porte le générateur.

Remarque:

Il n'est par contre pas indispensable que les paramètres formels homologues aient le même nom. Ainsi, le générateur de foncteur *fonction (pki)* est admissible bien que le nom *t* du paramètre formel de la fonction *pki* diffère du nom *x* du paramètre formel du type foncteur *fonction.* Il en va de même du générateur d'accesseur *charax (tabac).*

2. Au moyen d'un objet procédural anonyme. Un tel objet est noté au moyen d'un opérande **body**. Un tel opérande a l'une des formes suivantes:

body identificateur_de_type_acteur
paramètres_formels
declare suite_de_déclarations
do suite_d_énoncés **done**

body identificateur_de_type_foncteur
paramètres_formels
declare suite_de_déclarations
do suite_d_énoncés **take** expression **done**

body identificateur_de_type accesseur
paramètres_formels
declare suite_de_déclarations
do suite_d_énoncés **take** variable **done**

Un opérande **body** définit un objet procédural du type de l'identificateur placé après le symbole **body**. Il apparait clairement qu'un tel opérande constitue une sorte de bloc dans le sens qu'il introduit un nouveau niveau de déclarations incluant la partie formelle et la partie déclarative de l'opérande.

Remarques:

- Si la partie formelle d'un opérande **body** est vide, cet opérande hérite la partie formelle du type procédural concerné. Ceci apparait clairement dans la manière dont l'objet *rapcar* des exemples précédents a été construit. L'opérande **body** utilisé à cet effet est strictement équivalent à:

body *fonction (real* **value** *x)* **declare**
real **value** *x2 = x * * 2*
do take *(1 - x2) / (1 + x2)* **done**

- Si la partie formelle d'un opérande **body** n'est pas vide, il y aura compatibilité entre elle et la partie formelle du type procédural concerné (au sens où cette compatibilité a été définie plus haut).

Opérations sur les objets procéduraux.

Evaluation

L'élaboration de l'algorithme désigné par un objet procédural est réalisée au moyen d'une opération d'indiçage. On peut le constater, dans la série d'exemples précédents, dans la définition des objets *pint*, *partie_paire* et *pairapport*; ce dernier cas illustre comment on a pu fabriquer une nouvelle fonction par l'application de l'objet *partie_paire* à l'objet *rapport* (c'est-à-dire à la fonction *pki*). L'évaluation d'objets procéduraux sans paramètres est réalisée au moyen de l'opérateur postfixé **eval** que l'on peut aussi noter *[]*. Revenant aux exemples précédents, l'énoncé *pok* **eval** fera imprimer la chaîne *"<<< OK >>>"* tandis que *pal []* aura pour effet d'imprimer une valeur aléatoire.

Comparaison:

Deux objets procéduraux du même type peuvent être comparés pour l'égalité ou l'inégalité; ils seront considérés comme égaux s'ils dénotent le même algorithme exécuté dans le même environnement.

Exemple:

```
actor action variable a1, a2, a3;
procedure proc
      (integer value k)
declare
   procedure impr do print (line, k) done;
   action value a = action (impr)
do
   if k >= 0 then
      a1 := a; a2 := action (impr)
   default a3 := a; proc (-k) done
done (* proc *)
```

L'exécution de l'énoncé *proc (-12)* impliquera le stockage d'objets procéduraux dans les trois variables *a1*, *a2*, et *a3*. Ces objets dénoteront tous la procédure *impr*; on peut vérifier que l'on aura *a1 = a2*, mais par contre *a 1 ~= a3:* en-effet, l'exécution de *a3* aura lieu dans un autre environnement que celle de *a 1* ou *a2*. Dans le cas présent, cette différence apparaîtra clairement lorsque l'on exécute les algorithmes correspondants: *a1* **eval** ou *a2* **eval** feront imprimer la valeur *12*, mais *a3* **eval** fera imprimer la valeur *- 12*. Tout ceci apparait clairement dans le programme *compact* suivant:

```
compact                                                    Vax Newton Compiler 0.2
Page    1
                    Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]COMPACT.NEW*/ */
     1 PROGRAM compact DECLARE
     4   ACTOR action VARIABLE a1, a2, a3;
    13
    13   PROCEDURE faire(action VARIABLE a; PROCEDURE acte) DO
    24     a:=a1; acte;
    30     a:=a2; acte;
    36     a:=a3; acte
    41   DONE(*faire*);
    43
    43   PROCEDURE proc
    45      (integer VALUE k)
    50   DECLARE
    51     PROCEDURE impr DO print(line,k) DONE;
    62     action VALUE a=action(impr)
    70   DO
    71     IF k>=0 THEN
    76       a1:=a; a2:=action(impr)
    86     DEFAULT a3:=a; proc(-k) DONE
    97   DONE(*proc*)
    98 DO(*compact*)
    99   proc(-12);
   105   print(line,"***Evaluation des objets proceduraux***");
   112   faire(LAMBDA VALUE a, a EVAL);
   122   print(line,"<<<FIN>>>");
   129   print(line,"***Comparaison des objets proceduraux***");
   136   faire(LAMBDA VALUE a,
   142         (line; faire(LAMBDA VALUE b,print(a=b))));
   161   print(line,"<<<FIN>>>")
   167 DONE(*compact*)

     ****  No messages were issued  ****
```

Résultats:

```
***Evaluation des objets proceduraux***
         12
         12
        -12
<<<FIN>>>
***Comparaison des objets proceduraux***
  'TRUE'  'TRUE' 'FALSE'
  'TRUE'  'TRUE' 'FALSE'
 'FALSE' 'FALSE'  'TRUE'
<<<FIN>>>
```

On va voir maintenant différentes applications des objets procéduraux. Au moyen d'objets fonctionnels, on peut créer et manipuler des fonctions de nature relativement compliquée. A la limite, on peut en arriver à un style de programmation purement fonctionnel dans lequel une application est programmée définissant, de proche en proche, une fonction que l'on fait ensuite évaluer pour les données appropriées. Certains langages de programmation, orientés vers l'intelligence artificielle, encouragent un tel style de programmation: c'est en particulier le cas de Lisp.

Le programme *fonctions* suivant montre la construction de la fonction *cosexp[x] = cos (exp (x))* ainsi que de quelques fonctions qui en sont dérivées: sa deuxième, sa treizième et sa quatorzième réitération ainsi que sa partie paire. On peut remarquer la fonction *compose* qui permet d'exprimer la composition fonctionnelle de deux fonctions données, la fonction *réitère* qui exprime la réitération d'un rang donné d'une fonction, ainsi que l'objet *partie_paire* qui permet de livrer la partie paire d'une fonction donné.

```
fonctions                                              Vax Newton Compiler 0.2
Page    1
                       Source listing

/* /*OLDSOURCE=USER2:[RAPIN]FONCTIONS.NEW*/ */
PROGRAM fonctions DECLARE
  real FUNCTOR fonction(real VALUE x)VALUE
    identite=BODY fonction DO TAKE x DONE;

  fonction FUNCTION compose
    (fonction VALUE f,g)
  (*La composition des fonctions  f  et  g  donnees*)
  DO(*compose*)TAKE
    BODY
      fonction(real VALUE x)
    DO TAKE f[g[x]] DONE
  DONE(*compose*);

  fonction FUNCTION reitere
    (fonction VARIABLE f; integer VARIABLE k)
  (*La reiteration d'ordre  k  de la fonction  f .

    Condition d'emploi: k>=0
  *)
  DECLARE fonction VARIABLE r:=identite DO
    IF k<0 THEN
      print(line,"###Facteur de reiteration negatif###")
    RETURN DONE;
    WHILE k>0 REPEAT
      WHILE EVEN k REPEAT
        f:=compose(f,f); k:=k%2
      REPETITION;
      r:=compose(r,f); k:=PRED k
    REPETITION
  TAKE r DONE(*reitere*);

  fonction FUNCTOR fonctionnelle(fonction VALUE f)VALUE
    partie_paire=
      BODY
        fonctionnelle(fonction VALUE f)
      DO TAKE
        BODY
          fonction(real VALUE x)
        DO TAKE (f[x]+f[-x])/2 DONE
      DONE;
```

```
fonction VALUE
  cosinus=BODY fonction DO TAKE cos(x) DONE,
  exponentielle=BODY fonction DO TAKE exp(x) DONE,
  cosexp=compose(cosinus,exponentielle),
  cosexp_2=reitere(cosexp,2),
  cosexp_13=reitere(cosexp,13),
  cosexp_14=reitere(cosexp,14),
  paire_cosexp=partie_paire[cosexp]
DO(*fonctions*)
  FOR real VALUE x FROM -1 BY .05 TO 1.5 REPEAT
    line; edit(x,5,2);
          edit(cosexp[x],12,8); edit(cosexp_2[x],12,8);
          edit(cosexp_13[x],12,8); edit(cosexp_14[x],12,8);
          edit(paire_cosexp[x],12,8)
  REPETITION;
  print(line,"<<<FIN>>>")
DONE(*fonctions*)

   ****  No messages were issued  ****
```

Pour fabriquer la fonction *réitère*, on constate que l'on part de la fonction identique *identité* qui est l'élément neutre par rapport à la composition fonctionnelle. Dans la partie exécutable du programme, on constate que l'énoncé **for** peut porter sur un compteur de type réel: dans ce cas, le pas de tabulation doit obligatoirement être spécifié au moyen d'une clause **by**.

y = x;

f = cos(cos(x))

y = cos(cos(x))

f = cos cos cos cos +

y = cos cos cos cos cos cos +

f = 8 (cos)

y = 14 (cos)

Résultats:

-1.00	.93309208	-.82576763	.89416055	-.76721386	.01067908
-.95	.92614317	-.81571057	.89415410	-.76720376	.03835418
-.90	.91848279	-.80441485	.89414661	-.76719200	.07108131
-.85	.91004035	-.79172426	.89413784	-.76717825	.10736540
-.80	.90073878	-.77746470	.89412753	-.76716208	.14589105
-.75	.89049401	-.76144322	.89411533	-.76714294	.18552350
-.70	.87921453	-.74344747	.89410079	-.76712014	.22530107
-.65	.86680086	-.72324566	.89408334	-.76709276	.26442232
-.60	.85314506	-.70058747	.89406220	-.76705960	.30222998
-.55	.83813033	-.67520594	.89403635	-.76701905	.33819365
-.50	.82163055	-.64682106	.89400438	-.76696890	.37189222
-.45	.80350996	-.61514501	.89396434	-.76690608	.40299705
-.40	.78362288	-.57988986	.89391346	-.76682625	.43125623
-.35	.76181364	-.54077778	.89384770	-.76672305	.45648046
-.30	.73791660	-.49755433	.89376096	-.76658693	.47853052
-.25	.71175641	-.45000469	.89364384	-.76640307	.49730645
-.20	.68314866	-.39797300	.89348110	-.76614754	.51273837
-.15	.65190077	-.34138400	.89324692	-.76577969	.52477899
-.10	.61781343	-.28026627	.89289492	-.76522646	.53339757
-.05	.58068262	-.21477525	.89233538	-.76434630	.53857534
.00	.54030231	-.14521411	.89137766	-.76283762	.54030231
.05	.49646807	-.07204956	.88956227	-.75997043	.53857534
.10	.44898172	.00408030	.88556826	-.75362870	.53339757
.15	.39765722	.08236913	.87446055	-.73575801	.52477899
.20	.34232808	.16185889	.82851578	-.65872369	.51273837
.25	.28285648	.24147118	.51351671	-.10019312	.49730645
.30	.21914445	.32005280	-.37442939	.77271971	.47853052
.35	.15114727	.39643314	-.67845939	.87401174	.45648046
.40	.07888957	.46948862	-.73709405	.88768549	.43125623
.45	.00248414	.53820771	-.75382696	.89131350	.40299705
.50	-.07784610	.60174931	-.76027438	.89268053	.37189222
.55	-.16174304	.65948733	-.76327181	.89331032	.33819365
.60	-.24868510	.71103625	-.76484896	.89364023	.30222998
.65	-.33795622	.75625478	-.76575357	.89382901	.26442232
.70	-.42861239	.79522826	-.76630589	.89394411	.22530107
.75	-.51944702	.82823290	-.76665909	.89401765	.18552350
.80	-.60895668	.85568769	-.76689287	.89406630	.14589105
.85	-.69530956	.87809984	-.76705148	.89409929	.10736540
.90	-.77632017	.89600989	-.76716075	.89412201	.07108131
.95	-.84943482	.90994016	-.76723636	.89413773	.03835418
1.00	-.91173391	.92034803	-.76728802	.89414847	.01067908
1.05	-.95995871	.92758181	-.76732165	.89415546	-.01028233
1.10	-.99057181	.93183313	-.76734060	.89415940	-.02273186
1.15	-.99986222	.93307384	-.76734602	.89416053	-.02478711
1.20	-.98410682	.93095637	-.76733674	.89415860	-.01456196
1.25	-.93980050	.92464050	-.76730819	.89415267	.00971849
1.30	-.86396549	.91248144	-.76724934	.89414043	.04956350
1.35	-.75454693	.89146700	-.76713437	.89411653	.10601904
1.40	-.61089378	.85623193	-.76689704	.89406716	.17942748
1.45	-.43431211	.79746755	-.76633277	.89394971	.26915110
1.50	-.22865895	.69985644	-.76456615	.89358115	.37327531

<<<FIN>>>

Lorsque l'on examine les résultats de ce programme, on remarque que les treizième et quatorzièmes réitérations de la fonction *consexp* ont des comportements presque inverses. La première part d'une valeur proche de *0.89* pour décroitre brusquement, au voisinage de $x = 0.25$, vers une valeur proche de - *0.76*; la quatorzième réitération part, au contraire, d'une valeur proche de - *0.76* pour croitre brusquement vers *0.89*. Ces comportements peuvent

s'expliquer par la présence de points fixes: une fonction $f(x)$ a un point fixe en $x = a$ si $f(a) = a$. On remarque tout d'abord que $- 0.76$ et 0.89 ne sont pas des points fixes de la fonction *cosexp*, par contre, l'on a *cosexp [- 0.76] = 0.89* et *cosexp [0.89] = - 0.76* (environ). Il s'ensuit que ces deux valeurs $- 0.76$ et 0.89 seraient des points fixes de la deuxième réitération *cosexp_2* de la fonction *cosexp*; en inspectant les résultats, il apparait que cette fonction a effectivement des points fixes proches de ces deux valeurs: elle en a, de plus un troisième, proche de $x = 0.25$; ce dernier point fixe est aussi un point fixe de la fonction *cosexp* elle-même.

Un point fixe a peut être stable ou instable. On dira qu'il est stable si, lorsqu'on part d'une valeur $a[0]$ proche du point fixe a, la suite $a[k] = f(a[k - 1])$ converge vers a. Par contre, on dira que le point fixe a est instable si cette même suite diverge de a (elle peut même éventuellement converger vers un autre point fixe). Le comportement de ces différentes réitérations peut s'expliquer si l'on admet que $- 0.76$ et 0.89 sont des points fixes stables de la deuxième réitération *cosexp_2* tandis que le point fixe proche de 0.25 est instable.

On peut chercher à déterminer, de manière plus précise, ces points fixes: ceci implique la résolution numérique de l'équation *cosexp_2 [x] - x = 0*. Pour cela, on peut souvent recommander la schéma itératif de Newton lorsque la fonction que l'on cherche à annuler est dérivable et lorsque l'on connait une bonne approximation initiale de la solution. Soit $a[0]$ cette approximation, la suite $a[k] = a[k - 1] - f(a[k - 1]) / f'(a[k - 1])$ converge en général quadratiquement vers la solution a de l'équation $f(x) = 0$ si le point initial $a[0]$ est suffisamment proche de a.

Le programme *points_fixes* suivant montre la manière dont cette méthode peut être mise en oeuvre pour déterminer les trois points fixes de la fonction *cosexp_2*. On remarque qu'il n'a pas été nécessaire d'expliciter la formule donnant la dérivée de *cosexp_2*. L'idée est de laisser l'ordinateur construire cette dérivée sous la forme d'un objet fonctionnel. On l'a fait par l'intermédiaire des objets du type *fonc_c1*; ces objets ont deux fonctions *fonc* et *fonc_prime* comme composantes. Lors de la construction de chaque objet *fonc_c1*, on a fait en sorte que la fonction *fonc_prime* soit la dérivée de *fonc*. On peut constater, par exemple, la manière dont la règle de dérivation des fonctions composées a été appliquée pour construire l'objet résultant de l'application de la fonction *compose*.

```
points_fixes                                         Vax Newton Compiler 0.2
Page 1
                    Source listing

   /* /*OLDSOURCE=USER2:[RAPIN]POINTS_FIXES.NEW*/ */
   PROGRAM points_fixes DECLARE
     real FUNCTOR fonction(real VALUE x)VALUE
       identite=BODY fonction DO TAKE x DONE,
       un=BODY fonction DO TAKE 1 DONE;

     OBJECT fonc_c1(fonction VALUE fonc,fonc_prime)VALUE
       ident=fonc_c1(identite,un);

     fonc_c1 FUNCTION compose
       (fonc_c1 VALUE f,g)
     (*La composition des fonctions  f  et  g  donnees*)
     DO(*compose*)TAKE
       fonc_c1(BODY
                 fonction(real VALUE x)
               DO TAKE f.fonc[g.fonc[x]] DONE,
               BODY
                 fonction(real VALUE x)
               DO TAKE
                 f.fonc_prime[g.fonc[x]]*g.fonc_prime[x]
               DONE)
     DONE(*compose*);

     fonc_c1 FUNCTION moins
       (fonc_c1 VALUE f,g)
     (*La difference entre les fonctions  f  et  g  *)
     DO TAKE
       fonc_c1(BODY
                 fonction(real VALUE x)
               DO TAKE f.fonc[x]-g.fonc[x] DONE,
               BODY
                 fonction(real VALUE x)
               DO TAKE f.fonc_prime[x]-g.fonc_prime[x] DONE)
     DONE(*moins*);

     fonction VALUE
       neg_sinus=BODY fonction DO TAKE -sin(x) DONE,
       cosinus=BODY fonction DO TAKE cos(x) DONE,
       exponentielle=BODY fonction DO TAKE exp(x) DONE;

     fonc_c1 VALUE c=fonc_c1(cosinus,neg_sinus),
                   e=fonc_c1(exponentielle,exponentielle),
                   cosexp=compose(c,e),
                   cosexp_2=compose(cosexp,cosexp),
                   cosexp_2_moins_x=moins(cosexp_2,ident);

     PROCEDURE newton
       (fonc_c1 VALUE f; real VARIABLE x; real VALUE ec_max)
     (*Recherche un zero de la fonction  f  de classe C1 au
       voisinage de la valeur  x  en utilisant le schema
       iteratif de Newton. Arrete les calculs lorsque l'ecart
       absolu entre deux approximations successives ne
```

```
points_fixes                                        Vax Newton Compiler 0.2
Page    2
                    Source listing

 282      depasse pas  ec_max .
 282    *)
 282    DECLARE real VARIABLE fx,dfx,ec DO
 291      print(line,"***Recherche d'un zero au voisinage de:"_,
 298                 edit(x,15,10),"***");
 310      CONNECT f THEN
 313        UNTIL
 314          x:=x-(ec:=(fx:=fonc[x])/(dfx:=fonc_prime[x]));
 340          print(line,edit(x,14,10),
 353                     edit(fx,14,10),edit(dfx,14,10),ec)
 373        TAKE ABS ec<=ec_max REPETITION
 379      DONE;
 381      print(line,"***Zero trouve en:"_,edit(x,15,10),"***",line)
 401    DONE(*newton*)
 402  DO(*points_fixes*)
 403    newton(cosexp_2_moins_x,-.76,&-8);
 416    newton(cosexp_2_moins_x,.25,&-8);
 428    newton(cosexp_2_moins_x,.89,&-8)
 439  DONE(*points_fixes*)

      ****  No messages were issued  ****
```

Résultats:

```
***Recherche d'un zero au voisinage de:     -.7600000000***
  -.7671792223  -.0047983798  -.6683704151 +.71792223469414635&-02
  -.7671498793   .0000197732  -.6738639719 -.29343048021964860&-04
  -.7671498788   .0000000003  -.6738417002 -.48491727672318183&-09
***Zero trouve en:     -.7671498788***

***Recherche d'un zero au voisinage de:      .2500000000***
   .2645579336  -.0085288166   .5858535158 -.14557933563326641&-01
   .2646416668  -.0000484699   .5788613469 -.83733254838350019&-04
   .2646416701  -.0000000019   .5788153445 -.33266329295276814&-08
***Zero trouve en:      .2646416701***

***Recherche d'un zero au voisinage de:      .8900000000***
   .8941404166   .0027623203  -.6671599775 -.41404166182091909&-02
   .8941197514  -.0000139258  -.6738750424 +.20665187900805137&-04
   .8941197509  -.0000000003  -.6738417006 +.51125419652337448&-09
***Zero trouve en:      .8941197509***
```

Les résultats de ce programme sont typiques: ils illustrent la convergence quadratique du schéma itératif de Newton: à chaque itération, le nombre de chiffres significatifs corrects double approximativement.

Remarque:

Dans les deux programmes précédents, on peut constater que l'on n'a pas utilisé un générateur *fonction (cos)* pour fabriquer l'objet *cosinus*, mais un opérande **body** *fonction* **do take** *cos(x)* **done.** Ceci vient d'une restriction d'implantation du langage Newton: il n'est pas autorisé de faire porter un générateur d'objet procédural sur l'identificateur d'une procédure ou d'une fonction prédéfinie, même si cette dernière a le profil approprié. Cette restriction pourra éventuellement être levée par la suite.

Le programme *tris* suivant montre la manière dont des objets fonctionnels peuvent être utilisés pour réaliser une procédure de tri de rangée très générale. Cette procédure *trier* admet deux paramètres: le premier est la rangée que l'on souhaite trier; le second est la relation d'ordre selon laquelle le tri doit être accompli: cette relation d'ordre prend la forme d'un objet fonctionnel du type *relation.*

```
tris                                        Vax Newton Compiler 0.2
Page     1
                  Source listing


     1 /* /*OLDSOURCE=USER2:[RAPIN]TRIS.NEW*/ */
     1 PROGRAM tris DECLARE
     4   real ROW vecteur;
     8
     8   Boolean FUNCTOR relation(real VALUE gauche,droite)VALUE
    19     croissant=BODY relation DO TAKE gauche<droite DONE,
    30     decroissant=BODY relation DO TAKE gauche>droite DONE,
    41     absolu=BODY relation DO TAKE ABS gauche<ABS droite DONE,
    54     classes_entieres=
    56       BODY relation DO TAKE FLOOR gauche<FLOOR droite DONE;
    67
    67   PROCEDURE trier
    69     (vecteur VALUE tab; relation VALUE inf)
    78   (*Rearrange les composantes du vecteur donne  tab  de maniere
    78     telle que l'on ait:
    78
    78       LOW tab<=j/\j<k/\k<=HIGH tab IMPL ~inf[tab[K],tab[j]]
    78
    78     Conditions d'emploi:
    78
    78     inf  denote une relation d'ordre (eventuellement partielle)
    78          sur les valeur reelles; pour  x, y, z  reels, on doit
    78          avoir:
    78
    78     inf[x,y] NAND inf[y,x] ;
    78     inf[x,y] NOR inf[y,x] IMPL (inf[x,z]==inf[y,z]) ;
    78     inf[x,y]/\inf[y,z] IMPL inf[x,z]
    78   *)
    78   DECLARE(*trier*)
    79     PROCEDURE tri_sec(integer VALUE bas,haut)DECLARE
    89       real VALUE elt=tab[bas];
    98       integer VARIABLE b:=bas, h:=SUCC haut
   108     DO(*tri_sec*)
   109       (*Invariants:
   109         bas<=k/\k<=b IMPL ~inf[elt,tab[k]]
   109         h<=k/\k<=haut IMPL ~inf[tab[k],elt]
   109       *)
   109       CYCLE zig_zag REPEAT
   112         WHILE
   113           inf[elt,tab[(h:=PRED h)]]
   127         REPETITION;
   129       IF b=h EXIT zig_zag DONE;
   137         tab[b]:=:tab[h];
   147         WHILE
   148           inf[tab[(b:=SUCC b)],elt]
   162         REPETITION;
   164       IF b=h EXIT zig_zag DONE;
   172         tab[b]:=:tab[h]
   181       REPETITION;
   183       IF bas<(b:=PRED b) THEN tri_sec(bas,b) DONE;
   201       IF (h:=SUCC h)<haut THEN tri_sec(h,haut) DONE
   218     DONE(*tri_sec*)
```

```
tris                                              Vax Newton Compiler 0.2
Page      2
                    Source listing

    DO tri_sec(LOW tab,HIGH tab) DONE(*trier*);

    PROCEDURE imprimer(vecteur VALUE vec) DO
      THROUGH vec INDEX k VALUE vk REPEAT
        IF k\5=1 THEN line DONE; edit(vk,12,8)
      REPETITION
    DONE(*imprimer*);

    PROCEDURE traiter
      (integer VALUE taille; real EXPRESSION terme;
       relation VALUE ordre)
    DECLARE(*traiter*)
      vecteur VALUE vec=
        THROUGH vecteur(1 TO taille):=terme REPETITION;
    DO(*traiter*)
      print(page,"***Vecteur original***");
      imprimer(vec);
      trier(vec,ordre);
      print(line,"***Vecteur trie***");
      imprimer(vec); print(line,"<<<FIN>>>")
    DONE(*traiter*)
  DO(*tris*)
    randomize;
    traiter(100,random,croissant);
    traiter(100,random,decroissant);
    traiter(50,normal,absolu);
    traiter(50,10*poisson,classes_entieres)
  DONE(*tris*)

      ****  No messages were issued  ****
```

Ainsi, on remarque que ce programme a construit et trié:

- Une rangée de *100* valeurs aléatoires par ordre croissant.
- Une rangée de *100* valeurs aléatoires par ordre décroissant.
- Une rangée de *50* valeurs aléatoires, tirées selon la distribution normale, dans l'ordre des valeurs absolues croissantes.
- Une rangée de *50* valeurs aléatoires, tirées selon une distribution exponentielle négative de moyenne *10*, par classes entières; dans ce dernier tri, deux valeurs sont considérées comme équivalentes si elles ont la même partie entière: elles ne seront pas triées entre elles.

```
***Vecteur original***
    .16989565   .00882000   .63612592   .87440106   .97520414
    .92607838   .51996730   .21448974   .21456974   .73817980
    .43060801   .84253652   .00525199   .84577375   .84308745
    .48506620   .86918733   .89691272   .85957762   .18613616
    .26324198   .10205018   .24322742   .08531776   .35901470
    .85883543   .51816638   .83171014   .08096497   .78877363
    .95555098   .29820178   .34035279   .05472446   .85263914
    .69680111   .87284646   .50261065   .13934919   .25730933
    .85945198   .41508574   .25184385   .39391480   .11974299
    .37263829   .48322549   .16170931   .42586376   .86167638
    .01679414   .61860094   .31084757   .56057656   .39014457
    .26664092   .28703346   .15904888   .45083180   .62332844
    .81164834   .58559875   .94733758   .71379648   .52650099
    .92880933   .19258987   .67657317   .37021929   .89515237
    .03015820   .69884139   .76722469   .17483204   .18431948
    .40161647   .85407287   .15207874   .11943252   .01217531
    .98672762   .47694711   .78420601   .37865002   .34697520
    .33218766   .67596064   .14911822   .70698131   .32353616
    .71427274   .04810793   .15275257   .89538163   .61159562
    .34933629   .10558202   .12836314   .26841140   .42820329
***Vecteur trie***
    .00525199   .00882000   .01217531   .01679414   .03015820
    .04810793   .05472446   .08096497   .08531776   .10205018
    .10558202   .11943252   .11974299   .12836314   .13934919
    .14911822   .15207874   .15275257   .15904888   .16170931
    .16989565   .17483204   .18431948   .18613616   .19258987
    .21448974   .21456974   .24322742   .25184385   .25730933
    .26324198   .26664092   .26841140   .28703346   .29820178
    .31084757   .32353616   .33218766   .34035279   .34697520
    .34933629   .35901470   .37021929   .37263829   .37865002
    .39014457   .39391480   .40161647   .41508574   .42586376
    .42820329   .43060801   .45083180   .47694711   .48322549
    .48506620   .50261065   .51816638   .51996730   .52650099
    .56057656   .58559875   .61159562   .61860094   .62332844
    .63612592   .67596064   .67657317   .69680111   .69884139
    .70698131   .71379648   .71427274   .73817980   .76722469
    .78420601   .78877363   .81164834   .83171014   .84253652
    .84308745   .84577375   .85263914   .85407287   .85883543
    .85945198   .85957762   .86167638   .86918733   .87284646
    .87440106   .89515237   .89538163   .89691272   .92607838
    .92880933   .94733758   .95555098   .97520414   .98672762
<<<FIN>>>
```

```
***Vecteur original***
    .06251509    .37831448    .20380203    .03479686    .66256412
    .55869461    .38581806    .54682051    .30929418    .97949563
    .51767972    .04924952    .52969589    .65818071    .64338698
    .58115359    .89844101    .17515460    .36519573    .00372368
    .47388161    .33638430    .40437496    .24670317    .36376211
    .89717860    .76715159    .20026921    .09633074    .53199905
    .98527916    .00582322    .95410864    .31237899    .15446763
    .53435206    .35592955    .49153079    .63508014    .16862803
    .43060089    .46242657    .04278843    .52359766    .47850270
    .57229118    .81556771    .06895205    .98976339    .39516966
    .82770353    .02823132    .01633685    .06011151    .25080087
    .62741178    .98895933    .06712283    .64653614    .65778478
    .99724961    .16541091    .17519329    .16575831    .47423773
    .97869530    .85549769    .08237046    .51947938    .91877474
    .51740793    .75426403    .30526995    .72434041    .75977199
    .99008101    .29596286    .31362264    .98061745    .20644205
    .59021215    .58866465    .00765169    .13634757    .85846928
    .96933042    .11386760    .16193703    .31567046    .45387741
    .48765199    .65459048    .66977742    .65937716    .62675518
    .54890300    .61915773    .84872657    .84307487    .62709054
***Vecteur trie***
    .99724961    .99008101    .98976339    .98895933    .98527916
    .98061745    .97949563    .97869530    .96933042    .95410864
    .91877474    .89844101    .89717860    .85846928    .85549769
    .84872657    .84307487    .82770353    .81556771    .76715159
    .75977199    .75426403    .72434041    .66977742    .66256412
    .65937716    .65818071    .65778478    .65459048    .64653614
    .64338698    .63508014    .62741178    .62709054    .62675518
    .61915773    .59021215    .58866465    .58115359    .57229118
    .55869461    .54890300    .54682051    .53435206    .53199905
    .52969589    .52359766    .51947938    .51767972    .51740793
    .49153079    .48765199    .47850270    .47423773    .47388161
    .46242657    .45387741    .43060089    .40437496    .39516966
    .38581806    .37831448    .36519573    .36376211    .35592955
    .33638430    .31567046    .31362264    .31237899    .30929418
    .30526995    .29596286    .25080087    .24670317    .20644205
    .20380203    .20026921    .17519329    .17515460    .16862803
    .16575831    .16541091    .16193703    .15446763    .13634757
    .11386760    .09633074    .08237046    .06895205    .06712283
    .06251509    .06011151    .04924952    .04278843    .03479686
    .02823132    .01633685    .00765169    .00582322    .00372368
<<<FIN>>>
```

L'algorithme incorporé dans la procédure *trier* est le "Quick Sort" dû à Hoare. En général pour trier une rangée de *n* éléments, cet algorithme nécessite un nombre d'opérations proportionnel à $n * ln\ (n)$; dans certains cas défavorables, il peut cependant en exiger jusqu'à $n * * 2$.

```
***Vecteur original***
    .81568349   .32605740   .55047258   .56159925  1.42575477
   3.06944294  -.12571693   .92007618 -1.13193978   .67355222
   -.92294506  1.60978860  -.22924721  -.31745116  1.13677088
   -.36929940   .64997881   .00617001   .68814374  1.65712575
    .56525351  1.85026834 -1.85655209   .18092412  1.09685380
    .05544381  -.22744743  1.28880608   .52138735  -.35586411
    .21277032   .80517710  1.89624473  -.39814012  -.49784338
    .06114405 -1.92962972  -.15250631 -1.28437969  1.18462140
   -.09135495   .39111272  1.34841075  -.12529638  -.08289414
    .16349229  -.76580497  -.02827417  2.49451186   .31042224
***Vecteur trie***
    .00617001  -.02827417   .05544381   .06114405  -.08289414
   -.09135495  -.12529638  -.12571693  -.15250631   .16349229
    .18092412   .21277032  -.22744743  -.22924721   .31042224
   -.31745116   .32605740  -.35586411  -.36929940   .39111272
   -.39814012  -.49784338   .52138735   .55047258   .56159925
    .56525351   .64997881   .67355222   .68814374  -.76580497
    .80517710   .81568349   .92007618  -.92294506  1.09685380
  -1.13193978  1.13677088  1.18462140 -1.28437969  1.28880608
   1.34841075  1.42575477  1.60978860  1.65712575  1.85026834
  -1.85655209  1.89624473 -1.92962972  2.49451186  3.06944294
<<<FIN>>>
```

```
***Vecteur original***
   6.10101863 21.45822604 12.51421771  7.34137642  2.64401738
   4.88442209  8.19410300  6.23894629  5.78113202 10.27474195
   7.08988834 16.80430960 10.18912304  1.42340030  7.06033311
    .38053601  3.60541538   .34540185  3.28660802  2.17397357
   2.60690822  3.02680900  5.89853917  5.17096501 12.27981771
    .52061046  2.08721861  1.47998461   .68441590 19.61393175
  20.39049934  9.16942645   .03504589  5.20731921 24.02202037
   4.62637623 11.48986410 21.63719420  6.17392117  7.71847183
   6.77842360 10.45107254  7.04729851  2.22382373 16.59536209
   2.31836757 10.05451892  7.40716994   .82480010  1.26567778
***Vecteur trie***
    .38053601   .03504589   .68441590   .52061046   .34540185
    .82480010  1.47998461  1.42340030  1.26567778  2.17397357
   2.22382373  2.64401738  2.31836757  2.08721861  2.60690822
   3.28660802  3.02680900  3.60541538  4.62637623  4.88442209
   5.89853917  5.17096501  5.20731921  5.78113202  6.77842360
   6.17392117  6.10101863  6.23894629  7.34137642  7.04729851
   7.08988834  7.06033311  7.71847183  7.40716994  8.19410300
   9.16942645 10.27474195 10.45107254 10.18912304 10.05451892
  11.48986410 12.51421771 12.27981771 16.59536209 16.80430960
  19.61393175 20.39049934 21.45822604 21.63719420 24.02202037
<<<FIN>>>
```

L'idée consiste à extraire un élément de la rangée, par exemple le premier, puis de placer cet élément à sa place définitive en stockant avant lui les éléments qui le précéderont et après lui ceux qui le suivront. Ce placement est réalisé au moyen d'un parcours en zigzag dans lequel on part alternativement des deux bouts de la rangée; chaque parcours partiel est interrompu lorsque l'on trouve un élément mal placé: cet élément est alors échangé avec l'élément que l'on a extrait primitivement. Finalement, on trie récursivement (si nécessaire) la portion de la rangée qui précède et celle qui suit l'élément considéré. La figure 8 suivante illustre le zigzag dans la cas d'une rangée de huit éléments que l'on veut trier par ordre croissant: on admet que c'est la première composante, de valeur *6.5*, qui sera placée définitivement.

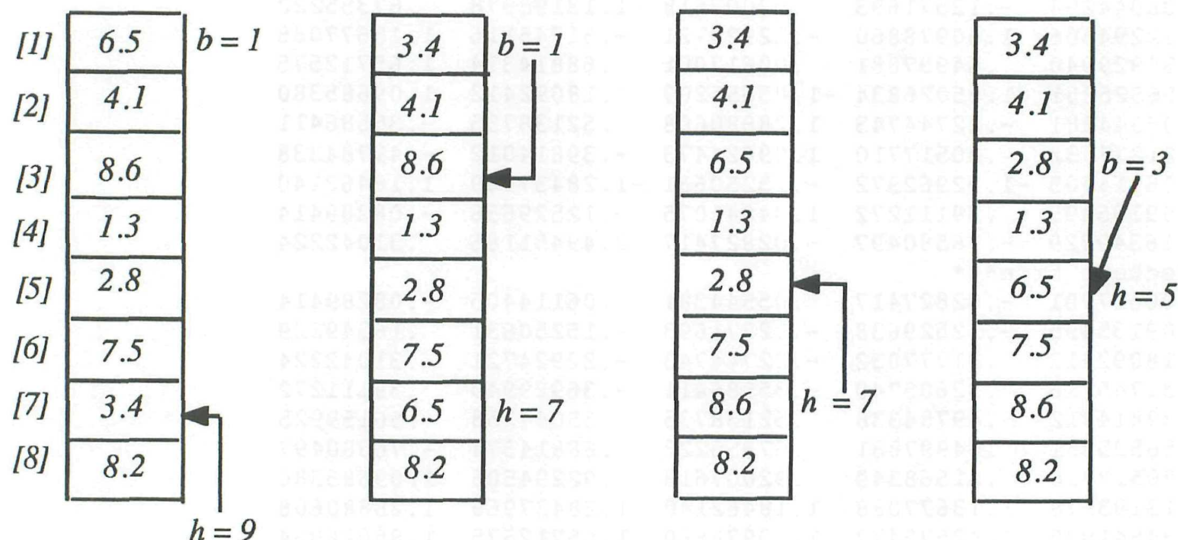


Figure 8

Après le dernier zigzag, on constate que les deux curseurs *b* et *h* se rencontrent à la position d'indice *5*: cette position est l'emplacement définitif de l'élément de valeur *6.5*; les tris récursifs portent ensuite sur les portions de la rangée situées entre les indices *1* et *4* d'une part, et celle située entre les indices *6* et *8* d'autre part.

Au moyen d'objets procéduraux, il est possible de faire coexister plusieurs représentations d'une même structure de donnée. L'idée consiste à découpler la déclaration du type abstrait (ou classe), qui décrit la structure de données et ses attributs, de son implantation. Plus spécifiquement, le type abstrait sera déclaré sous la forme d'une classe protocole; une implantation spécifique de ce type prendra la forme d'une fonction génératrice. Cette fonction livrera pour résultat une objet de type approprié; les opérations attributs seront insérées dans cet objet sous la forme d'objets procéduraux.

Le programme *matrix* suivant montre un premier exemple de cette technique. Ce programme va créer et manipuler des matrices carrées de formes diverses (symétriques, triangulaires droite et gauche en plus de matrices quelconques); dans le cas des matrices symétriques et triangulaires, on veut stocker uniquement la partie significative de la matrice.

```
matrix                                          Vax Newton Compiler 0.2
Page    1
                    Source listing

/* /*OLDSOURCE=USER2:[RAPIN]MATRIX.NEW*/ */
PROGRAM matrix DECLARE
  integer SUBRANGE positif
    (positif>0
     DEFAULT
       print(line,"###Entier non positif remplace par 1###")
     TAKE 1);
  real FUNCTOR constructeur(positif VALUE i,j);
  real ACCESSOR selecteur(positif VALUE i,j);

  SCALAR forme(ordinaire,symetrique,
               triangulaire_gauche,triangulaire_droite);

  CLASS matrice_carree
    INDEX element
    ATTRIBUTE genre,ordre
    (positif VALUE ordre; forme VALUE genre;
     selecteur VALUE elt)
  (*Un objet  matrice_carree  represente une matrice carree
    de dimension  ordre . La matrice sera implantee de telle
    sorte que la variable  elt[i,j]  implante l'element sur
    la  i_e  ligne et la  j_e  colonne de la matrice.
  *)
  DECLARE(*matrice_carree*)
    real ACCESS element
      (positif VALUE i,j)
    (*L'element d'indices  [i,j]  de la matrice*)
    DO(*element*)
      UNLESS i MAX j<=ordre THEN
        print(line,"###Erreur d'indicage###")
      RETURN DONE
    TAKE elt[i,j] DONE(*element*)
  DO(*matrice_carree*)DONE;
  /* /*EJECT*/ */
```

On remarque tout d'abord que les matrices carrées seront des objets de la classe protocole *matrice_carrée*. A chaque objet de cette classe, il sera passé comme paramètre un objet accesseur *elt* du type *sélecteur*; la fonction d'accès liée à cet objet devra implanter l'accès à *l'élément* spécifié par les indices passés en paramètres. Comme la fonction d'accès *élément* de la classe *matrice_carrée* fait le contrôle de validité des indices, celui qui implantera l'objet accesseur associé au paramètre *elt* n'a plus à s'en préoccuper.

```
matrix                                        Vax Newton Compiler 0.2
Page     2
                          Source listing

 matrice_carree FUNCTION matrice_generale
   (integer VALUE ordre; constructeur VALUE init)
 (*Resulte en une matrice carree de dimension  ordre ;
   l'element d'indices  [i,j]  aura la valeur initiale
   init[i,j] .
 *)
 DECLARE(*matrice_generale*)
   real ROW vecteur VALUE v=vecteur(1 TO ordre**2);

   real ACCESS sel
     (positif VALUE i,j)
   DO TAKE v[(i-1)*ordre+j] DONE
 DO(*matrice_generale*)
   FOR integer VALUE i FROM 1 TO ordre REPEAT
     FOR integer VALUE j FROM 1 TO ordre REPEAT
       sel(i,j):=init[i,j]
     REPETITION
   REPETITION
 TAKE(*matrice_generale*)
   matrice_carree(ordre,ordinaire,selecteur(sel))
 DONE(*matrice_generale*);

 matrice_carree FUNCTION matrice_symetrique
   (positif VALUE ordre; constructeur VALUE init)
 (*Resulte en une matrice symetrique de dimension  ordre ;
   l'element d'indices  [i,j]  avec  i>=j  aura pour
   valeur initiale  init[i,j] .
 *)
 DECLARE(*matrice_symetrique*)
   real ROW vecteur VALUE v=vecteur(1 TO ordre*(ordre+1)%2);

   real ACCESS elt
     (positif VALUE i,j)
   DECLARE
     positif VALUE ii=i MAX j,jj=i MIN j
   DO TAKE v[jj+ii*(ii-1)%2] DONE
 DO(*matrice_symetrique*)
   FOR integer VALUE i FROM 1 TO ordre REPEAT
     FOR integer VALUE j FROM 1 TO i REPEAT
       elt(i,j):=init[i,j]
     REPETITION
   REPETITION
 TAKE(*matrice_symetrique*)
   matrice_carree(ordre,symetrique,selecteur(elt))
 DONE(*matrice_symetrique*);
 /* /*EJECT*/ */
```

Il a ensuite été programmé quatre fonctions génératrices *matrice_générale*, *matrice_symétrique*, *matrice_gauche* et *matrice_droite* pour fournir des implantations des quatres formes de matrices considérées. Ces quatre fonctions ont été programmées selon les mêmes principes. La matrice considérée est implantée au moyen d'une rangée de dimension appropriée, avec une variable supplémentaire dans le cas des matrices triangulaires pour représenter l'ensemble des éléments nuls.

```
matrix                                          Vax Newton Compiler 0.2
Page    3
                    Source listing

matrice_carree FUNCTION matrice_gauche
  (positif VALUE ordre; constructeur VALUE init)
(*Resulte en une matrice triangulaire gauche de dimension
 ordre . L'element d'indices  [i,j]  avec  i>=j  aura pour
 valeur initiale  init[i,j] .
*)
DECLARE(*matrice_gauche*)
  real VARIABLE droite:=0;
  real ROW vecteur VALUE v=vecteur(1 TO ordre*(ordre+1)%2);

  real ACCESS elt
    (positif VALUE i,j)
  (*La composante d'indices  [i,j] *)
  DO(*elt*)TAKE
    IF i<j THEN
      droite:=0 TAKE droite
    DEFAULT v[j+i*(i-1)%2] DONE
  DONE(*elt*)
DO(*matrice_gauche*)
  FOR integer VALUE i FROM 1 TO ordre REPEAT
    FOR integer VALUE j FROM 1 TO i REPEAT
      elt(i,j):=init[i,j]
    REPETITION
  REPETITION
TAKE(*matrice_gauche*)
  matrice_carree(ordre,triangulaire_gauche,selecteur(elt))
DONE(*matrice_gauche*);


matrice_carree FUNCTION matrice_droite
  (positif VALUE ordre; constructeur VALUE init)
(*Resulte en une matrice triangulaire droite de dimension
  ordre . L'element d'indices  [i,j]  avec  i<=j  aura
  la valeur initiale  init[i,j] .
*)
DECLARE(*matrice_droite*)
  real ROW vecteur VALUE v=vecteur(1 TO ordre*(ordre+1)%2);
  real VARIABLE gauche:=0;

  real ACCESS elt
    (positif VALUE i,j)
  (*L'element d'indices  [i,j]  de la matrice *)
  DO(*elt*)TAKE
    IF i<=j THEN
      v[i+j*(j-1)%2]
    DEFAULT gauche:=0 TAKE gauche DONE
  DONE(*elt*)
DO(*matrice_droite*)
  FOR integer VALUE i FROM 1 TO ordre REPEAT
    FOR integer VALUE j FROM i TO ordre REPEAT
      elt(i,j):=init[i,j]
    REPETITION
  REPETITION
```

```
matrix                                          Vax Newton Compiler 0.2
Page    4
                      Source listing

  TAKE(*matrice_droite*)
    matrice_carree(ordre,triangulaire_droite,selecteur(elt))
  DONE(*matrice_droite*);
  /* /*EJECT*/ */
```

La fonction d'accès incorporée à chacune de ces fonctions génératrices montre les expressions utilisables pour implanter une matrice carrée, symétrique ou triangulaire, au moyen d'une rangée (d'un tableau) à un seul indice; c'est cette fonction d'accès transformée en un objet accesseur au moyen d'un générateur d'objet procédural, qui sera associée au paramètre *elt* de l'objet *matrice_carrée* résultant. On remarque que dans le cas des matrices triangulaires, on annule la variable représentative à chaque accès d'un des éléments identiquement nul: cette précaution est nécessaire puisque l'utilisateur peut y stocker une valeur non nulle au moyen d'une assignation à l'un de ces éléments.

Chacune de ces fonctions génératrices possède un paramètre *init* du type *constructeur*; les matrices seront initialisées lors de leur création par l'intermédiaire de l'objet foncteur associé au paramètre *init*.

```
real FUNCTION sigma
  (integer VARIABLE bas; integer VALUE haut;
   real EXPRESSION terme)
(* sigma(LAMBDA value i:=b,h,expr(i))  resulte dans la somme
  des valeurs de l'expression  expr(i)  pour  i=b,b+1,b+2,... h
*)
DECLARE real VARIABLE s:=0 DO
  WHILE bas<=haut REPEAT
    s:=s+terme; bas:=SUCC bas
  REPETITION
TAKE s DONE(*sigma*);

matrice_carree FUNCTION produit
  (matrice_carree VALUE a,b)
(*Le produit des matrices  a  et  b .

  Condition d'emploi: a.ordre=b.ordre
*)
DECLARE positif VALUE ordre=a.ordre DO
  UNLESS b.ordre=ordre THEN
    print(line,"###Matrices incompatibles###")
  RETURN DONE
TAKE(*produit*)
  IF
    a.genre=triangulaire_gauche/\b.genre=triangulaire_gauche
  THEN
    matrice_gauche(ordre,
                   BODY
                     constructeur(positif VALUE i,j)
                   DO TAKE
                     sigma(LAMBDA VALUE k:=j,i,a[i,k]*b[k,j])
                   DONE) ELSE
  IF
    a.genre=triangulaire_droite/\b.genre=triangulaire_droite
  THEN
    matrice_droite(ordre,
                   BODY
                     constructeur(positif VALUE i,j)
                   DO TAKE
                     sigma(LAMBDA VALUE k:=i,j,a[i,k]*b[k,j])
                   DONE)
  DEFAULT
    matrice_generale(ordre,
                     BODY
                       constructeur(positif VALUE i,j)
                     DO TAKE
                       sigma(LAMBDA VALUE k:=1,ordre,a[i,k]*b[k,j])
                     DONE)
  DONE
DONE(*produit*);
/* /*EJECT*/ */
```

```
matrix                                          Vax Newton Compiler 0.2
Page    6
                    Source listing

 PROCEDURE imprimer_matrice
   (matrice_carree VALUE m)
 DECLARE
   string VALUE m_genre=
     CASE m.genre WHEN
     ordinaire THEN "ordinaire"|
     symetrique THEN "symetrique"|
     triangulaire_gauche THEN "triangulaire gauche"|
     triangulaire_droite THEN "triangulaire droite"
     DEFAULT "de genre inconnu" DONE
 DO(*imprimer_matrice*)
   print(line,"***Matrice"_,m_genre,"***");
   FOR integer VALUE i FROM 1 TO m.ordre REPEAT
     line;
     FOR integer VALUE j FROM 1 TO m.ordre REPEAT
       edit(m[i,j],12,8);
       IF j\6=0 THEN line DONE
     REPETITION
   REPETITION;
   print(line,"<<<FIN>>>")
 DONE(*imprimer_matrice*);

 matrice_carree VALUE
   sym_1=matrice_symetrique
           (5,BODY constructeur DO TAKE (i+j)/(i*j) DONE),
   sym_2=matrice_symetrique
           (5,BODY constructeur DO TAKE 1/(i**2+j**2) DONE),
   gen_1=produit(sym_1,sym_2),
   gch_1=matrice_gauche
           (5,BODY constructeur DO TAKE (i-j+1)/(i+j) DONE),
   gch_2=matrice_gauche
           (5,BODY constructeur DO TAKE 1/(5-(i-j)) DONE),
   gch_3=produit(gch_1,gch_2),
   drt_1=matrice_droite
           (5,BODY constructeur DO TAKE (j-i+1)/(5-(j-i)) DONE),
   drt_2=matrice_droite
           (5,BODY constructeur DO TAKE 1/(j**2-i**2+1) DONE),
   drt_3=produit(drt_1,drt_2),
   gen_2=produit(gen_1,gch_3),
   gen_3=produit(gen_2,drt_3)
 /* /*EJECT*/ */
```

On remarque ensuite la fonction *sigma*; le premier paramètre effectif de cette fonction lui sera communiqué sous forme liée: cette fonction réalise une implantation informatique de l'opérateur de sommation $\sum$.

```
matrix                                        Vax Newton Compiler 0.2
Page     7
                     Source listing

DO(*matrix*)
  imprimer_matrice(sym_1); imprimer_matrice(sym_2);
  imprimer_matrice(gen_1); page;
  imprimer_matrice(gch_1); imprimer_matrice(gch_2);
  imprimer_matrice(gch_3); page;
  imprimer_matrice(drt_1); imprimer_matrice(drt_2);
  imprimer_matrice(drt_3); page;
  imprimer_matrice(gen_2); imprimer_matrice(gen_3)
DONE(*matrix*)

     ****  No messages were issued  ****
```

La fonction suivante *produit* implante le produit matriciel; on remarque simplement que l'on a tenu compte du fait que le produit de deux matrices triangulaires de même structure redonne une telle matrice.

Le reste du programme ne nécessite guère de commentaires. Dans la procédure *imprimer_matrice*, on remarque un énoncé **case** utilisé ici dans un contexte d'expression pruisque cet énoncé doit produire une chaîne comme résultat. D'une manière générale, une clause **case** est considérée comme une variante. Une telle variante a la forme:

> **case** expression **when**
> suite_de_cas

D'une manière générale, la suite de cas comporte un ou plusieurs cas séparés par le symbole | ; chaque cas a la forme:

> constantes_de_cas alternative

L'expression entre **case** et **when** doit livrer une valeur du type *integer*, *character* ou d'un type scalaire défini par l'utilisateur.

Les constantes de cas comportent une ou plusieurs constantes, du type de l'expression sur laquelle porte le **case**, séparées par des virgules.

Plusieurs constantes consécutives peuvent être spécifiées au moyen d'une clause de la forme **from** constante **to** constante.

Une variante **case** est satisfaite ssi le valeur de l'expression est égale à l'une des constantes de cas; le cas échéant l'alternative contenue dans le cas correspondant est élaborée. Il va de soi qu'une telle variante peut être combinée, dans un énoncé conditionnel complexe, avec d'autres formes de variantes. On considère, par exemple, la déclaration *integer* **variable** *i, j, k*; l'énoncé conditionnel suivant est tout à fait admissible:

> **if** *i* > *k* **then** *print ("i > k")* **else**
> **case** *j* **when**
> *1,5* **then** *print ("j = 1 ou j = 5")* |
> **from** *7* **to** *10* **then** *print ("j entre 7 et 10")* **else**
> **unless** *k* < *0* **then** *print ("k non négatif")*
> **default** *print ("k négatif")* **done**

Résultats:

```
***Matrice symetrique***
  2.00000000  1.50000000  1.33333333  1.25000000  1.20000000
  1.50000000  1.00000000   .83333333   .75000000   .70000000
  1.33333333   .83333333   .66666667   .58333333   .53333333
  1.25000000   .75000000   .58333333   .50000000   .45000000
  1.20000000   .70000000   .53333333   .45000000   .40000000
<<<FIN>>>
***Matrice symetrique***
   .50000000   .20000000   .10000000   .05882353   .03846154
   .20000000   .12500000   .07692308   .05000000   .03448276
   .10000000   .07692308   .05555556   .04000000   .02941176
   .05882353   .05000000   .04000000   .03125000   .02439024
   .03846154   .03448276   .02941176   .02439024   .02000000
<<<FIN>>>
***Matrice ordinaire***
  1.55301659   .79394341   .47475281   .31431118   .22235071
  1.10437406   .55074050   .32380761   .21207930   .14897755
   .95482655   .46967286   .27349254   .17800200   .12451984
   .88005279   .42913904   .24833501   .16096335   .11229098
   .83518854   .40481874   .23324049   .15074017   .10495366
<<<FIN>>>
```

```
***Matrice triangulaire gauche***
   .50000000   .00000000   .00000000   .00000000   .00000000
   .66666667   .25000000   .00000000   .00000000   .00000000
   .75000000   .40000000   .16666667   .00000000   .00000000
   .80000000   .50000000   .28571429   .12500000   .00000000
   .83333333   .57142857   .37500000   .22222222   .10000000
<<<FIN>>>
***Matrice triangulaire gauche***
   .20000000   .00000000   .00000000   .00000000   .00000000
   .25000000   .20000000   .00000000   .00000000   .00000000
   .33333333   .25000000   .20000000   .00000000   .00000000
   .50000000   .33333333   .25000000   .20000000   .00000000
  1.00000000   .50000000   .33333333   .25000000   .20000000
<<<FIN>>>
***Matrice triangulaire gauche***
   .10000000   .00000000   .00000000   .00000000   .00000000
   .19583333   .05000000   .00000000   .00000000   .00000000
   .30555556   .12166667   .03333333   .00000000   .00000000
   .44273810   .21309524   .08839286   .02500000   .00000000
   .64563492   .33210979   .16388889   .06944444   .02000000
<<<FIN>>>
```

```
***Matrice triangulaire droite***
   .20000000   .50000000  1.00000000  2.00000000  5.00000000
   .00000000   .20000000   .50000000  1.00000000  2.00000000
   .00000000   .00000000   .20000000   .50000000  1.00000000
   .00000000   .00000000   .00000000   .20000000   .50000000
   .00000000   .00000000   .00000000   .00000000   .20000000
<<<FIN>>>
***Matrice triangulaire droite***
  1.00000000   .25000000   .11111111   .06250000   .04000000
   .00000000  1.00000000   .16666667   .07692308   .04545455
   .00000000   .00000000  1.00000000   .12500000   .05882353
   .00000000   .00000000   .00000000  1.00000000   .10000000
   .00000000   .00000000   .00000000   .00000000  1.00000000
<<<FIN>>>
***Matrice triangulaire droite***
   .20000000   .55000000  1.10555556  2.17596154  5.28955080
   .00000000   .20000000   .53333333  1.07788462  2.13850267
   .00000000   .00000000   .20000000   .52500000  1.06176471
   .00000000   .00000000   .00000000   .20000000   .52000000
   .00000000   .00000000   .00000000   .00000000   .20000000
<<<FIN>>>
```

```
***Matrice ordinaire***
   .73856052   .23828182   .08004877   .02329880   .00444701
   .50731266   .16160361   .05395565   .01564765   .00297955
   .43023004   .13604420   .04525794   .01309726   .00249040
   .39168873   .12326450   .04090909   .01182207   .00224582
   .36856395   .11559668   .03829978   .01105695   .00209907
<<<FIN>>>
***Matrice ordinaire***
   .14771210   .45386465   .95961308  1.91060496  4.51421743
   .10146253   .31134269   .65784205  1.30953913  3.09506674
   .08604601   .26383536   .55725171  1.10918385  2.62201650
   .07833775   .24008170   .50695654  1.00900621  2.38549139
   .07371279   .22582951   .47677944   .94889963  2.24357632
<<<FIN>>>
```

Revenant à la technique utilisée dans le programme *matrix*, on constate qu'elle implique un développement de logiciel par couches. On suppose que le programme utilisateur contienne la déclaration suivante:

> *matrice_carrée* **value** *gen* =
> *matrice_générale (2,* **body** *constructeur* **do**
> **take** *(i + 3 * j) / 4* **done)**

La figure 9 montre schématiquement la structure de données issue de *gen*.

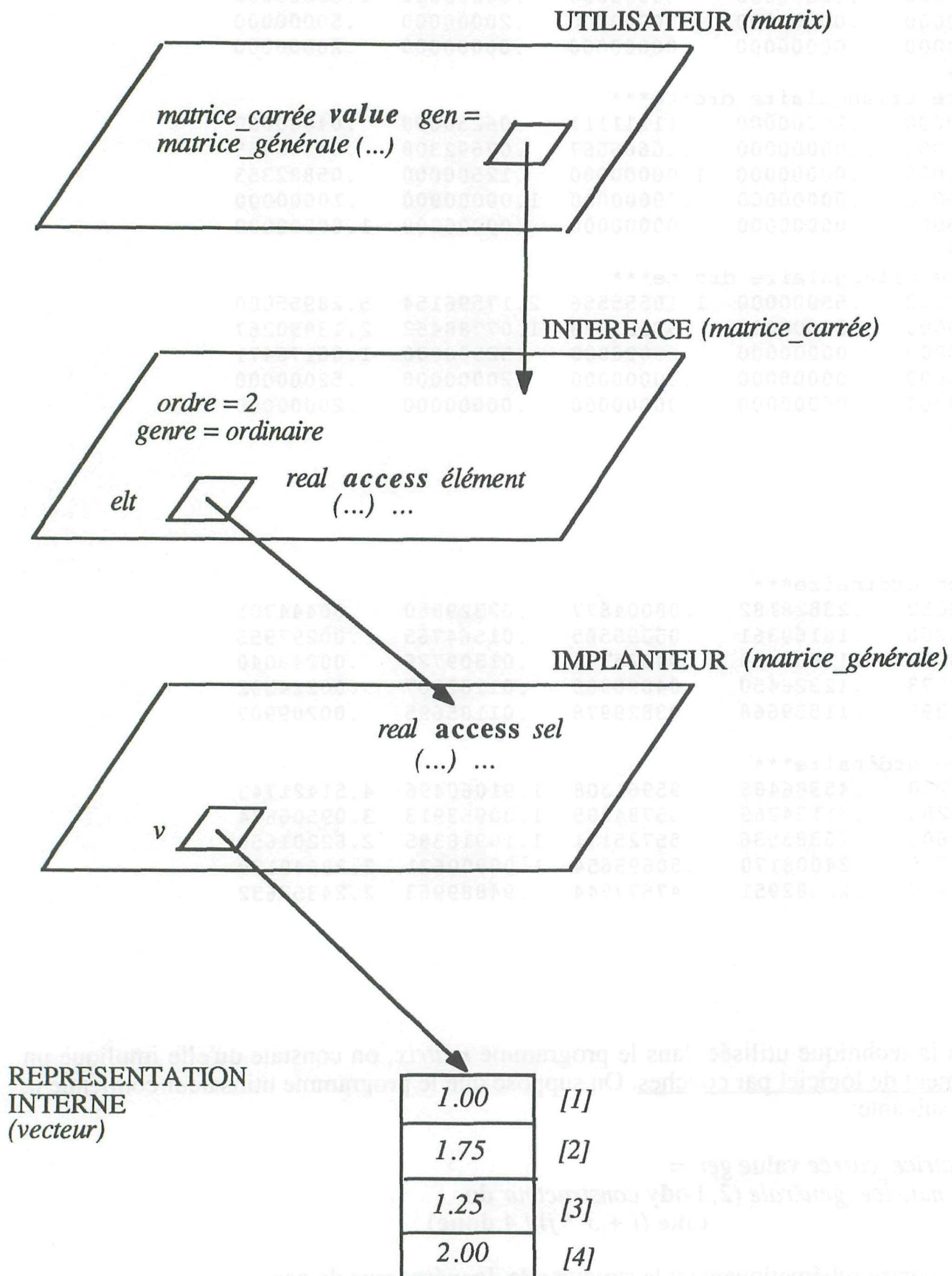

UTILISATEUR (matrix)
matrice_carrée value gen =
matrice_générale (...)
INTERFACE (matrice_carrée)
ordre = 2
genre = ordinaire
elt
real access élément
(...) ...
IMPLANTEUR (matrice_générale)
real access sel
(...) ...
v
REPRESENTATION
INTERNE
(vecteur)
1.00
[1]
1.75
[2]
1.25
[3]
2.00
[4]

Figure 9

- la couche supérieure est la couche utilisateur. On y trouve les entitées définies par ce dernier, en particulier la valeur *gen*.

- La couche suivante est une interface entre l'utilisateur et l'implanteur de la structure de données. L'utilisateur accède à cet interface par l'intermédiaire de la valeur *gen*, de ses attributs *gen . genre* et *gen . ordre* ainsi que de son opération d'indiçage *gen [i, j]*. Cette interface est l'objet produit par le générateur d'objet du type *matrice_carrée*.

- La couche inférieure est la couche implanteur. L'interface accède à cette couche par l'intermédiaire de l'objet accesseur *elt* (qui y dénote, par l'intermédiaire du générateur l'objet procédural *sélecteur (sel)*, la fonction d'accès *sel*) et de l'opération d'indiçage *elt [i, j]*. Cette couche implanteur est réalisée par le bloc d'activation de la fonction génératrice *matrice_générale*; ce bloc suivit à l'application de la fonction. Cette couche implanteur contient les entités déclarées dans la fonction génératrice, en particulier la fonction d'accès *sel* et la valeur *v*.

- Finalement, de la couche implanteur est issue la rangée représentative *v*. Cette rangée, sur laquelle porte la fonction d'accès *sel*, constitue la représentation interne de la structure de donnée.

Un autre intérêt des objets procéduraux est l'établissement de procédures ou fonctions mutuellement récursives. En Newton, la déclaration d'un identificateur doit textuellement precéder ses utilisations; ceci pose un problème si l'on doit établir un ensemble de procédures et/ou fonctions mutuellement récursives. Ce problème peut être résolu, de manière relativement simple, au moyen d'objets procéduraux. Le programme *murec* suivant montre le schéma de programmation que l'on peut adopter.

Ce programme incorpore une déclaration de module de la forme:

module identificateur
  attributs
declare
  suite_de_déclarations
do suite_d_énoncés **done**

```
murec                                                    Vax Newton Compiler 0.2
Page    1
                          Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]MUREC.NEW*/ */
     1 PROGRAM murec DECLARE
     4
     4 MODULE pq ATTRIBUTE p,q DECLARE
    11 (*Implante deux procedures  p  et  q  mutuellement recursives*)
    11   ACTOR acte(integer VALUE k)VARIABLE p_acte,q_acte;
    23
    23   PROCEDURE p(integer VALUE k)DO
    31     print(line,"P: k="_,k);
    41     IF k<0 THEN
    46       q_acte[-2*k]
    53     DEFAULT print(_"<<<FIN P>>>") DONE
    60   DONE(*p*);
    62
    62   PROCEDURE q(integer VALUE j)DO
    70     print(line,"Q: j="_,j);
    80     IF ABS j>50 THEN
    86       p_acte[-j%3]
    93     DEFAULT print(_"<<<FIN Q>>>") DONE
   100   DONE(*q*)
   101 DO(*pq*)
   102   p_acte:=acte(p); q_acte:=acte(q)
   115 DONE(*pq*)
   116
   116 DO(*murec*)
   117   print("***Essais de P***",line);
   124   p(35); line;
   131   p(-15); line;
   139   p(-35); print(line,"<<<FIN>>>");
   152   print(line,line,"***Essais de Q***",line);
   163   q(-125); line;
   171   q(25); line;
   178   q(125); print(line,"<<<FIN>>>")
   189 DONE(*murec*)

     ****  No messages were issued  ****
```

Un module peut être considéré comme un objet, construit à exemplaire unique au point de sa déclaration, et immédiatement connecté. Les attributs d'un module sont donc utilisables sans qualification dès la déclaration du module jusqu'à la fin du bloc contenant cette dernière. Dans le cas présent, le module *pq* exporte les deux procédures *p* et *q* mutuellement récursives.

Résultats:

```
***Essais de P***

P: k=          35 <<<FIN P>>>

P: k=         -15
Q: j=          30 <<<FIN Q>>>

P: k=         -35
Q: j=          70
P: k=         -23
Q: j=          46 <<<FIN Q>>>
<<<FIN>>>

***Essais de Q***

Q: j=        -125
P: k=          42 <<<FIN P>>>

Q: j=          25 <<<FIN Q>>>

Q: j=         125
P: k=         -41
Q: j=          82
P: k=         -27
Q: j=          54
P: k=         -18
Q: j=          36 <<<FIN Q>>>
<<<FIN>>>
```

On constate qu'à l'intérieur du module, y compris dans le corps de *p* et de *q*, on atteint ces procédures par l'intermédiaire des variables *p_acte* et respectivement *q_acte*; lors de l'initialisation du module, il est stocké dans ces variables les objets procéduraux *acte (p)* et *acte (q)* approprié.

Finalement, le programme *suites* suivant montre qu'il est tout à fait possible de définir des objets procéduraux récursifs. On y remarque notamment l'objet fonctionnel série qui transforme une suite dans la suite des sommes partielles de la série correspondante; on remarque que le résultat s livré par cet objet est, lui-même, un objet fonctionnel: ce dernier est défini récursivement.

```
suites                                          Vax Newton Compiler 0.2
Page    1
                         Source listing

/* /*OLDSOURCE=USER2:[RAPIN]SUITES.NEW*/ */ /* /*NO_XREF*/ */
PROGRAM suites DECLARE
  real FUNCTOR suite(integer VALUE n)
       FUNCTOR suite_transformation(suite VALUE s);

  suite_transformation VALUE serie=
    BODY
      suite_transformation(suite VALUE t)
    DECLARE
      suite VALUE s=
        BODY
          suite(integer VALUE n)
        DO TAKE
          IF n>0 THEN
            s[PRED n]+t[n]
          DEFAULT 0 DONE
        DONE
    DO TAKE s DONE;

  suite VALUE inv=BODY suite DO TAKE 1/n DONE,
              h=serie[inv],
              sh=serie[h],
              ssh=serie[sh]
DO(*suites*)
  FOR integer VALUE k FROM 1 TO 50 REPEAT
    line; edit(k,2,0); edit(inv[k],15,10); edit(h[k],15,10);
                       edit(sh[k],16,10); edit(ssh[k],17,10)
  REPETITION;
  print(line,"<<<FIN>>>")
DONE(*suites*)

    ****  No messages were issued  ****
```

On notera que vu le fort taux de récursion impliqué par l'élaboration d'entités telles que *ssh [50]*, l'exécution de ce programme est lente.

1	1.0000000000	1.0000000000	1.0000000000	1.0000000000
2	.5000000000	1.5000000000	2.5000000000	3.5000000000
3	.3333333333	1.8333333333	4.3333333333	7.8333333333
4	.2500000000	2.0833333333	6.4166666667	14.2500000000
5	.2000000000	2.2833333333	8.7000000000	22.9500000000
6	.1666666667	2.4500000000	11.1500000000	34.1000000000
7	.1428571429	2.5928571429	13.7428571429	47.8428571429
8	.1250000000	2.7178571429	16.4607142857	64.3035714286
9	.1111111111	2.8289682540	19.2896825397	83.5932539683
10	.1000000000	2.9289682540	22.2186507937	105.8119047619
11	.0909090909	3.0198773449	25.2385281385	131.0504329004
12	.0833333333	3.1032106782	28.3417388167	159.3921717172
13	.0769230769	3.1801337551	31.5218725719	190.9140442890
14	.0714285714	3.2515623266	34.7734348984	225.6874791875
15	.0666666667	3.3182289932	38.0916638917	263.7791430791
16	.0625000000	3.3807289932	41.4723928849	305.2515359640
17	.0588235294	3.4395525226	44.9119454075	350.1634813716
18	.0555555556	3.4951080782	48.4070534857	398.5705348573
19	.0526315789	3.5477396571	51.9547931429	450.5253280002
20	.0500000000	3.5977396571	55.5525328000	506.0778608002
21	.0476190476	3.6453587048	59.1978915048	565.2757523050
22	.0454545455	3.6908132502	62.8887047550	628.1644570600
23	.0434782609	3.7342915111	66.6229962661	694.7874533261
24	.0416666667	3.7759581778	70.3989544438	765.1864077699
25	.0400000000	3.8159581778	74.2149126216	839.4013203915
26	.0384615385	3.8544197162	78.0693323378	917.4706527293
27	.0370370370	3.8914567533	81.9607890911	999.4314418203
28	.0357142857	3.9271710390	85.8879601300	1085.3194019504
29	.0344827586	3.9616537976	89.8496139276	1175.1690158780
30	.0333333333	3.9949871309	93.8446010585	1269.0136169365
31	.0322580645	4.0272451954	97.8718462540	1366.8854631905
32	.0312500000	4.0584951954	101.9303414494	1468.8158046399
33	.0303030303	4.0887982257	106.0191396751	1574.8349443150
34	.0294117647	4.1182099904	110.1373496656	1684.9722939806
35	.0285714286	4.1467814190	114.2841310846	1799.2564250652
36	.0277777778	4.1745591968	118.4586902814	1917.7151153466
37	.0270270270	4.2015862238	122.6602765052	2040.3753918519
38	.0263157895	4.2279020133	126.8881785185	2167.2635703704
39	.0256410256	4.2535430389	131.1417215575	2298.4052919278
40	.0250000000	4.2785430389	135.4202645964	2433.8255565242
41	.0243902439	4.3029332828	139.7231978792	2573.5487544035
42	.0238095238	4.3267428066	144.0499406859	2717.5986950893
43	.0232558140	4.3499986206	148.3999393065	2865.9986343958
44	.0227272727	4.3727258933	152.7726651998	3018.7712995956
45	.0222222222	4.3949481156	157.1676133154	3175.9389129110
46	.0217391304	4.4166872460	161.5843005613	3337.5232134723
47	.0212765957	4.4379638417	166.0222644031	3503.5454778754
48	.0208333333	4.4587971751	170.4810615781	3674.0265394535
49	.0204081633	4.4792053383	174.9602669165	3848.9868063700
50	.0200000000	4.4992053383	179.4594722548	4028.4462786248

<<<FIN>>>

Chapitre 6

Traitement de texte

Trois types d'informations sont prédéfinis pour le traitement de texte:

character: une valeur du type *character* est un caractère individuel du jeu disponible sur l'ordinateur.

alphabet: une valeur du type *alphabet* est un sous-ensemble de caractères extrait du jeu disponible.

string: une valeur du type *string* est une suite ordonnée, une chaîne, de caractères de longueur finie mais non bornée à priori.

Le jeu de caractères disponibles sur le VAX est le jeu Ascii; ce jeu comporte *128* caractères dont *95* caractères imprimables (y compris l'espace blanc) et *33* caractères de contrôle (qui remplissent de fonctions spéciales).

Un caractère imprimable est noté en plaçant ce caractère entre des guillemets (sauf s'il s'agit d'un guillemets) ou entre des dièzes (sauf s'il s'agit d'un dièze); l'espace blanc est noté en plaçant un espace entre une paire de guillemets ou de dièzes ou alors au moyen du seul caractère de soulignage (non encadré de guillemets ou de dièzes).

Un caractère de contrôle est noté au moyen d'un mot-clé ad-hoc.

Exemples:

"A"	dénote le *A* majuscule
#A#	idem
"_"	dénote le caractère de soulignage
"#"	dénote le dièze
#"#	dénote le guillemet
# #	dénote l'espace blanc
_	idem
nul_char	dénote le caractère de contrôle vide (ce caractère est sans effet)
cr_char	dénote le retour de chariot au début de la ligne (Carriage Return)
lf_char	dénote l'avance à la ligne suivante (Line Feed).

Le programme *caractères* suivant indique la sémantique du type *character*. Il fait imprimer le jeu des caractères disponibles et montre, par l'exemple, la signification des opérations définies sur les caractères. Le lecteur constatera facilement que les caractères de contrôle ont été imprimés, dans les résultats, sous la forme de leur abréviation anglaise placée entre crochets pointus et non celle du mot clé qui les désigne en Newton: ainsi, le retour du chariot apparaît dans le résultats sous la forme *<CR>* et non **cr_char**. Les mots clés apparaissent, par contre, sous la forme de constantes de cas de la variante **case** de la procédure *print_car*; la signification des caractères de contrôle correspondants est consignée, sous la forme d'un commentaire, dans l'alternative correspondante.

```
caracteres                                        Vax Newton Compiler 0.2
Page    1
                         Source listing

/* /*OLDSOURCE=CARACTERES.NEW*/ */
PROGRAM caracteres DECLARE
(*Ce programme illustre la semantique du type predefini
  character .
*)
  CONSTANT printing_cars={FROM   TO #~#};

  PROCEDURE print_car(character VALUE c) DO
    IF c IN printing_cars THEN
      print(c)
    DEFAULT
      print("<"+
            string[TAKE CASE c WHEN
                   NUL_CHAR THEN "NUL"(*NULl character*)|
                   SOH_CHAR THEN "SOH"(*Start Of Heading*)|
                   STX_CHAR THEN "STX"(*Start of TeXt*)|
                   ETX_CHAR THEN "ETX"(*End of TeXt*)|
                   EOT_CHAR THEN "EOT"(*End Of Transmission*)|
                   ENQ_CHAR THEN "ENQ"(*ENQuiry*)|
                   ACK_CHAR THEN "ACK"(*ACKnowledge*)|
                   BEL_CHAR THEN "BEL"(*BELl*)|
                   BS_CHAR  THEN "BS"(*Back Space*)|
                   HT_CHAR  THEN "HT"(*Horizontal Tabulation*)|
                   LF_CHAR  THEN "LF"(*Line Feed*)|
                   VT_CHAR  THEN "VT"(*Vertical Tabulation*)|
                   FF_CHAR  THEN "FF"(*Form Feed*)|
                   CR_CHAR  THEN "CR"(*Carriage Return*)|
                   SO_CHAR  THEN "SO"(*Shift Out*)|
                   SI_CHAR  THEN "SI"(*Shift In*)|
                   DLE_CHAR THEN "DLE"(*Data Link Escape*)|
                   DC1_CHAR THEN "DC1"(*Device Control 1*)|
                   DC2_CHAR THEN "DC2"(*Device Control 2*)|
                   DC3_CHAR THEN "DC3"(*Device Control 3*)|
                   DC4_CHAR THEN "DC4"(*Device Control 4*)|
                   NAK_CHAR THEN "NAK"(*Negative AcKnowledge*)|
                   SYN_CHAR THEN "SYN"(*SYNchronous idle*)|
                   ETB_CHAR THEN "ETB"(*End of Transmission Block*)|
                   CAN_CHAR THEN "CAN"(*CANcel*)|
                   EM_CHAR  THEN "EM"(*End of Medium*)|
                   SUB_CHAR THEN "SUB"(*SUBstitute*)|
                   ESC_CHAR THEN "ESC"(*ESCape*)|
                   FS_CHAR  THEN "FS"(*File Separator*)|
                   GS_CHAR  THEN "GS"(*Group Separator*)|
                   RS_CHAR  THEN "RS"(*Record Separator*)|
                   US_CHAR  THEN "US"(*Unit Separator*)|
                   DEL_CHAR THEN "DEL"(*Delete*)
                   DONE]+">")
    DONE
  DONE(*print_car*);
  /* /*EJECT*/ */
```

```
caracteres                                       Vax Newton Compiler 0.2
Page    2
                       Source listing


   PROCEDURE print_chaine(string VALUE s)DECLARE
     character VARIABLE cur_del:=#"#,alt_del:="#"
   DO(*print_chaine*)
     print(cur_del);
     THROUGH s INDEX k VALUE c REPEAT
       IF k\16=0 THEN
         print(cur_del,line,__,cur_del)
       DONE;
       IF c=_ THEN
         print(cur_del," _ ",cur_del) ELSE
       IF c=cur_del THEN
         print(cur_del,_);
         cur_del:=:alt_del;
         print(cur_del); print_car(c)
       DEFAULT print_car(c) DONE
     REPETITION;
     print(cur_del)
   DONE(*print_chaine*);

   PROCEDURE print_alpha(alphabet VALUE a)DECLARE
     string VARIABLE s:=""
   DO(*print_alpha*)
     print("{");
     THROUGH a VALUE c REPEAT
       s:=s+c
     REPETITION;
     print_chaine(s);
     print("}")
   DONE(*print_alpha*);

   PROCEDURE operation
     (string VALUE op; character VARIABLE cc,dd; PROCEDURE impr)
   DO(*operation*)
     line; line; column(9-LENGTH op); print(op); column(11);
     THROUGH "abcdef" VALUE d REPEAT
       space(7); print(d)
     REPETITION;
     line;
     THROUGH "abcdef" VALUE c REPEAT
       line; column(8); print((cc:=c));
       THROUGH "abcdef" INDEX k VALUE d REPEAT
         dd:=d; column(8*k+3); impr
       REPETITION
     REPETITION
   DONE(*operation*)
/* /*EJECT*/ */
```

```
caracteres                                          Vax Newton Compiler 0.2
Page    3
                    Source listing

  427 DO(*caracteres*)
  428   print("1. Jeu lie a la calculatrice:",line,
  434         "    ==========================" ,line);
  439   FOR character VALUE c FROM character MIN TO character MAX REPEAT
  450     IF ORD c\16=0 THEN line DONE; print_car(c)
  465   REPETITION;
  467   page;
  469   print("2. Operateurs monadiques:",line,
  475         "    =====================" ,line);
  480   print(line,"c",column(12),"ORD c",
  493              column(18),"PRED c",column(26),"SUCC c",
  507              column(34),"UPCASE c",column(44),"LOWCASE c",line);
  524   FOR character VALUE c REPEAT
  529     line; print_car(c); column(9); edit(ORD c,8,0);
  551           column(18);
  556           UNLESS c=character MIN THEN
  562             print_car(PRED c)
  567           DEFAULT print("???") DONE;
  574           column(26);
  579           UNLESS c=character MAX THEN
  585             print_car(SUCC c)
  590           DEFAULT print("???") DONE;
  597           column(34);
  602           print_car(UPCASE c);
  608           column(44);
  613           print_car(LOWCASE c);
  619     IF ORD c\32=31 THEN page DONE
  629   REPETITION;
  631   print("3. Operateurs d'optimisation:",line,
  637         "    ========================" ,line);
  642   operation("c MIN d",LAMBDA VALUE c,LAMBDA VALUE d,
  654                       (space(7); print_car(c MIN d)));
  669   operation("c MAX d",LAMBDA VALUE c,LAMBDA VALUE d,
  681                       (space(7); print_car(c MAX d)));
  696   page;
  698   print("4. Operateurs de comparaison:",line,
  704         "    ========================", line);
  709   operation("c=d",LAMBDA VALUE c,LAMBDA VALUE d,print(c=d));
  729   operation("c~=d",LAMBDA VALUE c,LAMBDA VALUE d,print(c~=d));
  749   page;
  751   operation("c<d",LAMBDA VALUE c,LAMBDA VALUE d,print(c<d));
  771   operation("c<=d",LAMBDA VALUE c,LAMBDA VALUE d,print(c<=d));
  791   operation("c>d",LAMBDA VALUE c,LAMBDA VALUE d,print(c>d));
  811   operation("c>=d",LAMBDA VALUE c,LAMBDA VALUE d,print(c>=d));
  831   page;
  833   print("5. Forcages:",line,
  839         "    ========", line);
  844   print(line,"c",column(5),print("string[c]"),
  860                  column(15),print("alphabet[c]"),line);
  873   FOR character VALUE c FROM "a" TO "f" REPEAT
  882     line; print(c); column(5); print_chaine(string[c]);
  902                     column(15); print_alpha(alphabet[c])
```

```
caracteres                                          Vax Newton Compiler 0.2
Page    4
                    Source listing
```

```
  REPETITION;
  print(line,"<<<FIN>>>")
DONE(*caracteres*)

     ****  No messages were issued  ****
```

Résultats:

```
1. Jeu lie a la calculatrice:
   ==========================

<NUL><SOH><STX><ETX><EOT><ENQ><ACK><BEL><BS><HT><LF><VT><FF><CR><SO><SI>
<DLE><DC1><DC2><DC3><DC4><NAK><SYN><ETB><CAN><EM><SUB><ESC><FS><GS><RS><US>
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~<DEL>
```

```
2. Operateurs monadiques:
   ====================

c              ORD  c PRED  c  SUCC  c  UPCASE  c  LOWCASE c

<NUL>            0 ???        <SOH>     <NUL>       <NUL>
<SOH>            1 <NUL>      <STX>     <SOH>       <SOH>
<STX>            2 <SOH>      <ETX>     <STX>       <STX>
<ETX>            3 <STX>      <EOT>     <ETX>       <ETX>
<EOT>            4 <ETX>      <ENQ>     <EOT>       <EOT>
<ENQ>            5 <EOT>      <ACK>     <ENQ>       <ENQ>
<ACK>            6 <ENQ>      <BEL>     <ACK>       <ACK>
<BEL>            7 <ACK>      <BS>      <BEL>       <BEL>
<BS>             8 <BEL>      <HT>      <BS>        <BS>
<HT>             9 <BS>       <LF>      <HT>        <HT>
<LF>            10 <HT>       <VT>      <LF>        <LF>
<VT>            11 <LF>       <FF>      <VT>        <VT>
<FF>            12 <VT>       <CR>      <FF>        <FF>
<CR>            13 <FF>       <SO>      <CR>        <CR>
<SO>            14 <CR>       <SI>      <SO>        <SO>
<SI>            15 <SO>       <DLE>     <SI>        <SI>
<DLE>           16 <SI>       <DC1>     <DLE>       <DLE>
<DC1>           17 <DLE>      <DC2>     <DC1>       <DC1>
<DC2>           18 <DC1>      <DC3>     <DC2>       <DC2>
<DC3>           19 <DC2>      <DC4>     <DC3>       <DC3>
<DC4>           20 <DC3>      <NAK>     <DC4>       <DC4>
<NAK>           21 <DC4>      <SYN>     <NAK>       <NAK>
<SYN>           22 <NAK>      <ETB>     <SYN>       <SYN>
<ETB>           23 <SYN>      <CAN>     <ETB>       <ETB>
<CAN>           24 <ETB>      <EM>      <CAN>       <CAN>
<EM>            25 <CAN>      <SUB>     <EM>        <EM>
<SUB>           26 <EM>       <ESC>     <SUB>       <SUB>
<ESC>           27 <SUB>      <FS>      <ESC>       <ESC>
<FS>            28 <ESC>      <GS>      <FS>        <FS>
<GS>            29 <FS>       <RS>      <GS>        <GS>
<RS>            30 <GS>       <US>      <RS>        <RS>
<US>            31 <RS>                 <US>        <US>
```

	32	<US>	!		
!	33		"	!	!
"	34	!	#	"	"
#	35	"	$	#	#
$	36	#	%	$	$
%	37	$	&	%	%
&	38	%	'	&	&
'	39	&	(	'	'
(	40	'	)	(	(
)	41	(	*	)	)
*	42	)	+	*	*
+	43	*	,	+	+
,	44	+	-	,	,
-	45	,	.	-	-
.	46	-	/	.	.
/	47	.	0	/	/
0	48	/	1	0	0
1	49	0	2	1	1
2	50	1	3	2	2
3	51	2	4	3	3
4	52	3	5	4	4
5	53	4	6	5	5
6	54	5	7	6	6
7	55	6	8	7	7
8	56	7	9	8	8
9	57	8	:	9	9
:	58	9	;	:	:
;	59	:	<	;	;
<	60	;	=	<	<
=	61	<	>	=	=
>	62	=	?	>	>
?	63	>	@	?	?

@	64 ?	A	@	@
A	65 @	B	A	a
B	66 A	C	B	b
C	67 B	D	C	c
D	68 C	E	D	d
E	69 D	F	E	e
F	70 E	G	F	f
G	71 F	H	G	g
H	72 G	I	H	h
I	73 H	J	I	i
J	74 I	K	J	j
K	75 J	L	K	k
L	76 K	M	L	l
M	77 L	N	M	m
N	78 M	O	N	n
O	79 N	P	O	o
P	80 O	Q	P	p
Q	81 P	R	Q	q
R	82 Q	S	R	r
S	83 R	T	S	s
T	84 S	U	T	t
U	85 T	V	U	u
V	86 U	W	V	v
W	87 V	X	W	w
X	88 W	Y	X	x
Y	89 X	Z	Y	y
Z	90 Y	[	Z	z
[	91 Z	\	[	[
\	92 [	]	\	\
]	93 \	^	]	]
^	94]	_	^	^
_	95 ^	`	_	_

<table>
<tr><td>`</td><td>96</td><td>_</td><td>a</td><td>`</td><td>`</td></tr>
<tr><td>a</td><td>97</td><td>`</td><td>b</td><td>A</td><td>a</td></tr>
<tr><td>b</td><td>98</td><td>a</td><td>c</td><td>B</td><td>b</td></tr>
<tr><td>c</td><td>99</td><td>b</td><td>d</td><td>C</td><td>c</td></tr>
<tr><td>d</td><td>100</td><td>c</td><td>e</td><td>D</td><td>d</td></tr>
<tr><td>e</td><td>101</td><td>d</td><td>f</td><td>E</td><td>e</td></tr>
<tr><td>f</td><td>102</td><td>e</td><td>g</td><td>F</td><td>f</td></tr>
<tr><td>g</td><td>103</td><td>f</td><td>h</td><td>G</td><td>g</td></tr>
<tr><td>h</td><td>104</td><td>g</td><td>i</td><td>H</td><td>h</td></tr>
<tr><td>i</td><td>105</td><td>h</td><td>j</td><td>I</td><td>i</td></tr>
<tr><td>j</td><td>106</td><td>i</td><td>k</td><td>J</td><td>j</td></tr>
<tr><td>k</td><td>107</td><td>j</td><td>l</td><td>K</td><td>k</td></tr>
<tr><td>l</td><td>108</td><td>k</td><td>m</td><td>L</td><td>l</td></tr>
<tr><td>m</td><td>109</td><td>l</td><td>n</td><td>M</td><td>m</td></tr>
<tr><td>n</td><td>110</td><td>m</td><td>o</td><td>N</td><td>n</td></tr>
<tr><td>o</td><td>111</td><td>n</td><td>p</td><td>O</td><td>o</td></tr>
<tr><td>p</td><td>112</td><td>o</td><td>q</td><td>P</td><td>p</td></tr>
<tr><td>q</td><td>113</td><td>p</td><td>r</td><td>Q</td><td>q</td></tr>
<tr><td>r</td><td>114</td><td>q</td><td>s</td><td>R</td><td>r</td></tr>
<tr><td>s</td><td>115</td><td>r</td><td>t</td><td>S</td><td>s</td></tr>
<tr><td>t</td><td>116</td><td>s</td><td>u</td><td>T</td><td>t</td></tr>
<tr><td>u</td><td>117</td><td>t</td><td>v</td><td>U</td><td>u</td></tr>
<tr><td>v</td><td>118</td><td>u</td><td>w</td><td>V</td><td>v</td></tr>
<tr><td>w</td><td>119</td><td>v</td><td>x</td><td>W</td><td>w</td></tr>
<tr><td>x</td><td>120</td><td>w</td><td>y</td><td>X</td><td>x</td></tr>
<tr><td>y</td><td>121</td><td>x</td><td>z</td><td>Y</td><td>y</td></tr>
<tr><td>z</td><td>122</td><td>y</td><td>{</td><td>Z</td><td>z</td></tr>
<tr><td>{</td><td>123</td><td>z</td><td>|</td><td>{</td><td>{</td></tr>
<tr><td>|</td><td>124</td><td>{</td><td>}</td><td>|</td><td>|</td></tr>
<tr><td>}</td><td>125</td><td>|</td><td>~</td><td>}</td><td>}</td></tr>
<tr><td>~</td><td>126</td><td>}</td><td><DEL></td><td>~</td><td>~</td></tr>
<tr><td><DEL></td><td>127</td><td>~</td><td>???</td><td><DEL></td><td><DEL></td></tr>
</table>

3. Operateurs d'optimisation:
========================

c MIN d	a	b	c	d	e	f
a	a	a	a	a	a	a
b	a	b	b	b	b	b
c	a	b	c	c	c	c
d	a	b	c	d	d	d
e	a	b	c	d	e	e
f	a	b	c	d	e	f

c MAX d	a	b	c	d	e	f
a	a	b	c	d	e	f
b	b	b	c	d	e	f
c	c	c	c	d	e	f
d	d	d	d	d	e	f
e	e	e	e	e	e	f
f	f	f	f	f	f	f

4. Operateurs de comparaison:
========================

c=d	a	b	c	d	e	f
a	'TRUE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
b	'FALSE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
c	'FALSE'	'FALSE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'
d	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'FALSE'	'FALSE'
e	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'FALSE'
f	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'TRUE'

c~=d	a	b	c	d	e	f
a	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
b	'TRUE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
c	'TRUE'	'TRUE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'
d	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'TRUE'	'TRUE'
e	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'TRUE'
f	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'

c<d	a	b	c	d	e	f
a	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
b	'FALSE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
c	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'
d	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'TRUE'
e	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'TRUE'
f	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'

c<=d	a	b	c	d	e	f
a	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
b	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'
c	'FALSE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'

d	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'TRUE'	'TRUE'
e	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'TRUE'
f	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'TRUE'

c>d	a	b	c	d	e	f
a	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
b	'TRUE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
c	'TRUE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
d	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'
e	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'FALSE'
f	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'

c>=d	a	b	c	d	e	f
a	'TRUE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
b	'TRUE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
c	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'FALSE'	'FALSE'
d	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'	'FALSE'
e	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'FALSE'
f	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'	'TRUE'

```
5. Forcages:
   ========

c    string[c] alphabet[c]

a    "a"        {"a"}
b    "b"        {"b"}
c    "c"        {"c"}
d    "d"        {"d"}
e    "e"        {"e"}
f    "f"        {"f"}
<<<FIN>>>
```

Pour l'essentiel, les opérations disponibles sur les caractères sont les mêmes que celles qui le sont sur les types scalaires définis par l'utilisateur. On a déjà vu quelques exemples de tels types; leur déclaration à la forme:

scalar identificateur *(*suite_de_constantes_scalaires*)*

La suite de constantes scalaires prend la forme d'une liste d'identificateurs, séparés par des virgules. On peut considérer que le type prédéfini *character* est un type scalaire dont les valeurs sont notées au moyen d'une syntaxe spéciale et non d'identificateurs. Par exception, les opérateurs **upcase** et **lowcase**, ainsi que le forçage au type *string*, sont spécifiquement liées au traitement de texte: ces notions n'ont pas d'équivalents dans le cas des types scalaires.

Pour le reste, les résultats de ce programme ne nécessitent que peu de commentaires. Les opérateurs ont été classés dans différentes catégories de priotités décroissante. Ainsi, l'expression **pred** *"E"* **max** *"M"* signifie *(***pred** *"E")* **max** *"M"* et non **pred** *("E"* **max** *"M")*. On notera, et ceci est valable aussi pour les types scalaires, que l'inverse de l'opérateur **ord** est exprimable au moyen d'un forceur.

Exemples:

scalar *couleur (rouge, orange, jaune, vert, bleu, violet)*

Les expressions suivantes sont toutes vraies:

ord *[vert] = 3*
couleur [3] = vert
ord *["A"] = 65*
character [65] = "A"

Les expressions **pred nul_char** et **succ del_char** ne sont pas définies; leur "résultat" a été imprimé sous la forme *???*. On remarquera enfin que les valeurs extrêmes d'un type scalaire ou du type *character* peuvent toujours être notées en faisant suivre l'identificateur de ce type d'un des symboles **min** ou **max**.

Exemples:

couleur **min** = *rouge*
couleur **max** = *violet*
character **min** = **nul_char**
character **max** = **del_char**

Dans ces exemples, on a évidemment présupposé le même type *couleur*; on le supposera encore dans plusieurs exemples subséquents.

Le type prédéfini *alphabet* est un type ensemble, basé sur le type *character*. D'une manière générale, un type ensemble est déclaré de la manière suivante.

indication_de_type **set** identificateur

L'identifidation de type placée avant **set** doit nécessairement dénoter un type scalaire ou le type *character*; en particulier, le type *integer* (ou même un de ses sous-types) n'est pas autorisé.

Exemples:

character **set** *alphabet;*
couleur **set** *teinte*

La première de ces deux déclarations est prédéfinie. En général, les primitives disponibles sur le type *alphabet* le sont aussi sur les autres types ensemble: les exceptions seront signalées explicitement.

On peut former des ensembles constants: pour cela, on fait suivre l'identificateur du type ensemble considéré d'une suite de constantes du type de base, séparées par des virgules, et placées entre accolades.

Exemples:

alphabet {"A", "B", "C", "D", "E"}
teinte {rouge, vert, bleu}

Plusieurs constantes consécutives peuvent être incluses, dans un ensemble, au moyen d'une clause **from** constante **to** constante.

Exemples:

*alphabet {***from** *"A"* **to** *"E"}*
*teinte {***from** *rouge* **to** *jaune}*

L'ensemble vide et l'ensemble universel d'un type considéré peut être noté en faisant suivre l'identificateur de ce type d'un des symboles **empty** ou **full**; dans le cas de l'ensemble vide, il est possible d'utiliser *{}* au lieu de **empty**.

Exemples:

alphabet {} = alphabet **empty**
alphabet **full** = *alphabet {***from nul_char to del_char***}*
teinte **empty** = *teinte {}*
teinte {couleur **min to** *couleur* **max***}* = *teinte* **full**

Dans le cas du type *alphabet*, l'indicateur de ce type peut être omis devant la liste de constantes entre accolades (mais non devant **empty** ou **full**); de plus, la liste de constantes peut comporter des notations de chaînes.

Exemples:

alphabet {"A", "B", "C", "D", "E", "Z"}
alphabet {"ABCDEZ"}
*alphabet {***from** *"A"* **to** *"E", "Z"}*
{"A", "B", "C", "D", "E", "Z"}
*{***from** *"A"* **to** *"E", "Z"}*
{}

Les cinq premières constantes dénotent le même alphabet tandis que la dernière est égale à l'alphabet vide.
Il est possible de former des générateurs d'ensemble d'une syntaxe proche de celle des constantes d'ensemble. Un tel générateur permet de construire des valeurs d'un type ensemble à partir de valeurs du type de base calculées à l'exécution du programme. Par rapport à une constante d'ensemble, on a les différences suivantes:

- L'identificateur du type ensemble est suivi d'une liste parenthésée (au moyen de parenthèses rondes et non d'accolades); cet identificateur ne peut être omis dans les cas du type *alphabet*.
- En lieu et place de constantes, cette liste parenthésée peut contenir des expressions arbitraires susceptibles de livrer une valeur du type sur lequel est basé l'ensemble. Dans le cas de générateurs d'alphabets, ces expressions appartiendront uniquement au type *character* (le type *string* n'est pas autorisé: on notera, par contre, qu'une valeur du type *string* peut être forcée au type *alphabet*).

Un itérateur prédéfini est disponible sur les types ensemble; cet itérateur prend la forme:

through
  expression_d_ensemble
  value identificateur
repeat séquence_d_énoncés **repetition**

Dans la procédure *print_alpha* du programme *caractères* figure un exemple de cet itérateur:

through *a* **value** *c* **repeat**
  s := s + c
repetition

L'effet est d'exécuter la séquence d'énoncés entre **repeat** et **repetition** pour chacun des membres de l'ensemble; l'identificateur après le symbole **value** dénote le membre en question. Les membres sont traités dans l'ordre croissant de leurs valeurs. Utilisé comme expression, cet itérateur livre pour résultat l'ensemble sur lequel il a porté.

On va passer en revue les opérations disponibles sur les ensembles et, en particulier, sur les alphabets. Dans les groupes d'opérations suivants, *e* et *f* désigneront des valeurs du même type ensemble, *a* une valeur du type *alphabet* et *b* une valeur du type de base (type scalaire ou type *character*) du type ensemble de *e*.

Groupe 1. Opérateurs multiplicatifs monadiques.

card *e* : Le nombre de membres de *e*
Exemples: **card** *{} = 0*
card *teinte {rouge, jaune} = 2*
card *alphabet* **full** = *128*

upcase *a* : L'alphabet *a* dont toutes les minuscules ont été remplacées par les majuscules correspondantes.
Exemples: **upcase** *{"aBg."} = {"ABG."}*
upcase *{"Aaron"} = {"ANOR"}*

lowcase a : L'alphabet *a* dont toutes les majuscules ont été remplacées par les minuscules correpsondantes.
Exemples: **lowcase** *{"aBg."} = {"abg."}*
lowcase *{"Aaron"} = {"anor"}*
Remarque: Il est évident que les opérateurs **lowcase** et **upcase** ne sont applicables qu'aux ensembles du type *alphabet.*

Groupe 2. Opérateur multiplicatif dyadique

*e * f* : Intersection des ensembles concernés.
*e * b*
Exemples: *{"Tartine"} * {"Patate"} = {"ate"}*
*teinte {rouge, bleu, vert} * bleu = teinte {bleu}*

Groupe 3. Opérateur additif monadique.

- e : Complémentation de l'ensemble *e*
Exemple: *- teinte {rouge, jaune} =*
teinte {orange, vert, bleu, violet}

Groupe 4. Opérateurs additifs dyadiques

e + f : Réunion des ensembles concernés:
e + b
Exemples: *{"Tartine"} + {"Patate"} = {"PTaeinrt"}*
teinte {rouge, bleu} + vert =
teinte {rouge, vert, bleu}

e - f : Différences des ensembles concernés:
e - b
Exemples: *{"Tartine"} - {"Patate"} = {"Tinr"}*
teinte {rouge, bleu, jaune} - jaune =
teinte {rouge, bleu}

Groupe 5. Opérateur d'optimisation monadique

min *e* : L'élément de valeur minimum de *e*;
non défini sur l'ensemble vide.
Exemples: **min** *{"truc"} = "c"*
min *teinte {orange, vert} = orange*

max *e* : L'élément de valeur maximum de *e*;
non défini sur l'ensemble vide.
Exemple: **max** *{"truc"} = "u"*
max *teinte {orange, vert} = vert*

Groupe 6. Opérateurs interrogatifs

empty *e* : Vrai ssi *e* est l'ensemble vide.
Exemples: **empty** *{}*
~ **empty** *teinte {rouge}*

full *e* : Vrai ssi *e* est l'ensemble universel du type condidéré.
Exemples: ~ **full** *{***from** *"A"* to *"Z"}*
full *teinte* **full**

Groupe 7. Opérateurs d'appartenance.

b **in** *e* : Vrai ssi *b* est un membre de *e*.
Exemples: *"a"* **in** *{"Tartine"}*
jaune **in** *teinte* **full**

b **notin** *e* : Vrai ssi *b* n'est pas un membre de *e*
Exemples: *"x"* **notin** *{"Tartine"}*
vert **notin** *teinte {}*

Groupe 8. Comparaisons

e = *f* : Vrai ssi les ensembles *e* et *f* sont égaux.
Exemple: *{"Patate"} = {"Peta"}*

e ~= *f* : Vrai ssi les ensembles *e* et *f* sont différents.
Exemple: *{"Patate"} ~= {"Tartine"}*

e ><*f* : Vrai ssi les ensembles *e* et *f* sont disjoints.
Exemples: *{"Patate"} >< {"Vin"}*
teinte {rouge} >< *teinte {}*

e <>*f* : vrai ssi les ensembles *e* et *f* ne sont pas disjoints.
Exemples: *("Patate")* <> *{"Tartine"}*
teinte {rouge} <> *teinte {rouge}*

e <= *f* : Vrai ssi l'ensemble *e* est inclus dans *f*
Exemples: *{"Tarte"}* <= *{"Tartine"}*
teinte {rouge} <= *teinte {rouge}*

e < f : Vrai ssi *e* est strictement inclus dans *f* (*f* doit avoir un membre non inclus dans *e*)
Exemples: *{"Tarte"} < {"Tartine"}*
~ teinte {rouge} < teinte {rouge}

e >= f : vrai ssi *f* est inclus dans *e*

e > f : vrai ssi *f* est strictement inclus dans *e*

Le type d'information principal pour le traitement de texte est le type *string*; contrairement aux types *character* et *alphabet*, ce type n'a pas d'équivalent basé sur les types scalaires.

Une constante de chaîne peut être notée de l'une des manières suivantes:

- Une suite de caractères imprimables, ne comportant aucun guillemets, encadrée par des guillemets.
- Une suite de caractères imprimables, ne comportant aucun dièze, encadrée par des dièzes.
- La juxtaposition de deux ou plusieurs constantes de chaîne et/ou de caractères.

Cette troisième convention permet de former des constantes de chaîne contenant à la fois des dièzes et des guillemets, des contantes de chaînes contenant des caractères de contrôle ou des constantes de chaîne très longues (nécessitant plusieurs lignes de texte).

Exemples:

"" La chaîne vide

idem

#Il dit: "Voici une sonate en ut# "#"_#mineur!"#

"Cette chaîne est terminée par un caractère vide" **nul_char**

La procédure *print_chaîne* du programme *caractères* comporte un itérateur prédéfini sur les chaînes.
Cet itérateur a la forme:

through
  expression_de_chaîne
  index identificateur
  value identificateur
repeat suite_d_énoncés **repetition**

La suite d'énoncés incluse entre **repeat** et **repetition** est exécutée une fois pour chacun des caractères successifs de la chaîne sur laquelle porte l'itérateur. L'identificateur de la clause **value** y dénote le caractère courant; l'identificateur de la clause **index** y dénote la position du caractère courant (*1* pour le premier caractère de la chaîne, *2* pour le second et ainsi de suite). Utilisé comme expression, cet itérateur livre la chaîne sur laquelle il a opéré.

Remarque importante:

> Une chaîne ne doit pas être considérée comme un tableau (une rangée) de caractères. Une chaîne est une valeur pure. De-même qu'il n'y a pas, à priori, de raisons de considérer une valeur entière comme un tableau de chiffres, il n'y a pas plus de raison de traiter une chaîne comme un tableau de caractères.

Les opérations disponibles sur les chaînes sont largement une conséquence de cette remarque. Il est certes possible d'extraire un caractère individuel ou une sous-chaîne d'une chaîne en fonction de sa position dans cette dernière; il est souvent plus naturel de le faire en fonction d'un certain contexte.

Exemple:

Réaliser une fonction répondant au protocole suivant:

```
string function cond_subst
   (string value sujet, objet, texte)
(* Remplace, dans la chaîne sujet, l'occurence la plus gauche de la chaîne objet par la
   chaîne texte. Si sujet ne contient pas objet comme sous-chaîne, le résultat est la
   chaîne sujet non modifiée.
*)
              /* ... */
```

Ainsi, l'expression *cond_subtst ("Le grand père et la grand'mère", "grand", "petit")* aura pour résultat de la chaîne *"Le petit père et la grand'mère"*.

Une première méthode consiste à chercher, dans la chaîne *sujet* , la position de l'occurence concernée de la chaîne de la chaîne *objet* puis de réaliser la substitution. Cette méthode est évidemment lourde:

```
string function cond_subst
   (string value sujet, objet, texte)
declare
   integer variable pos := 0
do (* cond_subst*) take
   cycle recherche repeat
      if
         (succ pos =: pos) + length objet > length sujet
      exit recherche take sujet done;
      if
         sujet @ pos .. length objet = objet
      exit recherche take
         sujet .. (pred pos) + texte + sujet @ (pos + length objet)
      done
   repetition
done (* cond_substr *)
```

Dans cette formulation, on constate la manière dont un énoncé **cycle** peut produire un résultat lorsque le contexte l'exige: la clause **exit** doit être suivie d'une clause **take** expression; l'expression dit livrer une valeur (ou une variable) du type approprié. Pour le reste, on remarque qu'il a été nécessaire de se préoccuper de nombreux petits détails. La situation serait encore plus désagréable si les chaînes devaient être stockées dans des tableaux à bornes fixées dès la compilation. Utilisant des opérateurs qui portent sur les contextes, il est possible de reformuler la fonction *cond_subst* de la manière suivante:

```
string function cond_subst
   (string value sujet, objet, texte)
do (*cond_subst *) take
   if objet in sujet then
      sujet leftof objet + texte + sujet leftcut objet
   default sujet done
done (*cond_subst *)
```

Dans cette nouvelle fomrulation, *objet* est un contexte que l'on cherche dans la chaîne *sujet* au moyen de l'opération *objet* **in** *sujet*; si le contexte est trouvé, l'expression partielle *sujet* **leftof** *objet* est la sous-chaîne de *sujet* à gauche de la première occurence du contexte *objet* et l'expression *sujet* **leftcut** *objet* est la sous-chaîne de *sujet* à droite de cette même occurence.

Un contexte peut être indiqué au moyen d'un caractère, d'une chaîne ou d'un alphabet.

- Soit *s* une chaîne et *c* un caractère; *s* contient le contexte *c* ssi la chaîne contient une occurence, au moins, du caractère *c*.

- Soient *s* et *t* deux chaînes; *s* contient le contexte *t* ssi *t* est une sous-châine de *s*.
  Remarque: La chaîne vide est un contexte de n'importe quelle chaîne *s*. Son occurence la plus à gauche précède le premier caractère de *s;* son ocurence la plus à droite suit son dernier caractère.

- Soit *s* une chaîne et *a* un alphabet; *s* contient le contexte *a* ssi *s* contient une occurence (au moins) de l'un des caractères membres de *a*.
  Remarque: L'alphabet vide n'est contexte d'aucune chaîne.

Les expressions *c* **in** *s*, *t* **in** *s* et *a*<>*s* sont vraies ssi la chaîne *s* contient le contexte indiqué par le caractère *c*, la chaîne *t* et respectivement l'alphabet *a*; les relations contraires sont notées *c* **notin** *s*, *t* **notin** *s* et *a* >< *s*,

Exemples:

"a" **in** *"Patate"*
"ta" **in** *"Patate*
"tete" **notin** *"Patate"*
{"tete"} <> *"Patate"*
{"truc"} <> *"Patate"*

On va maintenent passer en revue les opérations disponibles sur les chaînes; on désignera par *s* et *t* des chaînes, par *c* un caractère, par *a* un alphabet, par *cta* un contexte spécifié au moyen d'un caractère, d'une chaîne ou d'un alphabet, par *ca* un contexte spécifié au moyen d'un caractère ou d'un alphabet exclusivement, par *ct* un contexte spécifié au moyen d'un caractère ou d'une chaîne exclusivement et par *k* une valeur entière.

Groupe 1. Opérateurs monadiques

length *s* : le nombre de caractères de la chaîne *s*.
Exemples: **length** *"Patate" = 6*
length *"" = 0*

upcase *s* : la chaîne *s* dans laquelle toutes les minuscules sont remplacées per les majuscules correspondantes.
Exemples: **upcase** *"Patate" = "PATATE"*
upcase *"Vient-il?" = "VIENT-IL?"*

lowcase *s* : la chaîne *s* dans laquelle toutes les majuscules sont remplacées par les minuscules correspondantes.
Exemples: **lowcase** *"Patate" = "patate"*
lowcase *"VIENT-IL?" = "vient-il?"*

Groupe 2. Sélecteurs

s **char** *k* : le *k_e* caractère de *s*
Conditions d'emploi:
1 <= k /\ k <= **length** *s*
Exemple: *"Patate"* **char** *3 = "t"*

s **leftocc** *a* : le caractère de *a* le plus à gauche dans *s*.
Condition d'emploi: *a < > s*
Exemples: *"Patate"* **leftocc** *{"at"}= "a"*
"Tartine" **leftocc** *{"eint"} = "t"*

s **rightocc** *a* : le caractère de *a* le plus à droite dans *s*.
Condition d'emploi: *a < > s*
Exemples: *"Patate"* **rightocc** *{"at"} = "t"*
"Tartine" **rightocc** *{"eint"} = "e"*

s **leftpos** *cta:* la position, dans *s*, de l'occurence la plus à gauche du contexte *cta*; si *cta* **notin** *s*, le résultat est *1* + **length** *s*
Exemples: *"Patate"* **leftpos** *"a" = 2*
"Patate" **leftpos** *{"at"} = 2*
"Patate" **leftpos** *"tat" = 3*
"Patate" **leftpos** *"tata" = 7*

s **rightpos** *cta:* la position, dans *s*, de l'occurence la plus à droite du contexte *cta*, si *cta* **notin** *s*, le résultat est *0* (si le contexte est un caractère ou un alphabet) ou *1* - **length** *t* s'il s'agit d'une chaîne.
Exemples: *"Patate"* **rightpos** *"a" = 4*
"Patate" **rightpos** *{"at"} = 5*
"Patate" **rightpos** *"tat" = 3*
"Patate" **rightpos** *"tata" = - 3*

Groupes 3. Opérateurs multiplicatifs

s * *k* : Réplication d'ordre *k* de *s*
Condition d'emploi: *k* > = *0*
Exemples: *"ta" * 3 = "tatata"*
*"truc" * 0 = ""*

s .. *k* : la sous-chaîne formée de *k* premiers caractères de *s*
Conditions d'emploi:
0 < = *k* /\ *k* < = **length** *s*
Exemples: *"Patate" .. 3 = "Pat"*
"Truc" .. 0 = ""

s @ *k* : la sous-chaîne de *s* débutant à son *k_e* caractère.
Conditions d'emploi:
1 < = *k* /\ *k* < = 1 + **length** *s*
Exemples: *"Patate" @ 3 = "tate"*
"Patate" @ 7 = ""

s **leftof** *cta*: la partie de *s* à gauche de la première occurence du contexte *cta*; si *cta* **notin** *s*, le résultat est *s*.
Exemples: *"Patate"* **leftof** *"t" = "Pa"*
"Patate" **leftof** *"te" = "Pata"*
"Patate" **leftof** *{"a", "t"} = "P"*
"Patate" **leftof** *"tata" = "Patate"*

s **atleft** *cta*: la partie de *s* débutant à l'occurence la plus à gauche du contexte *cta*; si *cta* **notin** *s*, le résultat est la chaîne vide.
Exemples: *"Patate"* **atleft** *"t" = "tate"*
"Patate" **atleft** *"te" = "te"*
"Patate" **atleft** *{"a", "t"} = "atate"*
"Patate" **atleft** *"tata" = ""*
Remarque: Pour toute chaîne *s* et pour tout contexte *cta*, les opérations *s* **leftof** *cta* et *s* **atleft** *cta* réalisent une coupure de *s;* en concaténant les deux morceaux on a: *s* **leftof** *cta* + *s* **atleft** *cta* = *s*

s **toleft** *cta*: La partie de *s* se terminant à l'occurence la plus à gauche du contexte *cta*; si *cta;* **notin** *s*, le résultat est la chaîne *s*.

s **leftcut** *cta*: la sous-chaîne de *s* débutant après l'occurence la plus à gauche du contexte *cta*, si *cta* **notin** *s*, le résultat est la chaîne vide.
Remarque: Ces deux derniers opérateurs réalisent aussi une coupure:
s **toleft** *cta* + *s* **leftcut** *cta* = *s*
Exemples: *"Patate"* **toleft** *{"a", "t"} = "Pa"*
"Patate" **leftcut** *{"a", "t"} = "tate"*
"Patate" **toleft** *"tata" = "Patate"*
"Patate" **lefcut** *"tata" = ""*

s **rightcut** *cta*: la partie de *s* qui précède l'occurence la plus à droite du contexte *cta*; si si *cta* **notin** *s*, le résultat est la chaîne vide.

s atright *cta*: la partie de *s* qui débute à l'occurence la plus à droite du contexte *cta*; si *cta* **notion** *s*, le résultat est *s*.
Remarque: On a encore une coupure:
s rigthcut *cta* + **s atright** *cta* = *s*
Exemples: *"Patate"* **rightcut** *{"a","t"}* = *"Pata"*
"Patate" **atright** *{"a", "t"}* = *"te"*
"Patate" **rightcut** *"tata"* = *""*
"Patate" **atright** *"tata"* = *"Patate"*

s toright *cta*: la sous-chaîne de *s* se terminant à l'occurence la plus à droite du contexte *cta;* si *cta* **notin** *s*, le résultat est la chaîne vide.

s **rightof** *cta*: la sous-chaîne de *s* débuttant après la dernière occurence du contexte *cta*; si *cta* **notin** *s*, le résultat est *s*.
Remarque: on a encore une coupure:
s toright *cta* + **s righof** *cta* = *s*
Exemples: *"Patate"* **toright** *{"a", "t"}* = *"Patat"*
"Patate" **rightof** *{"a", "t"}* = *"e"*
"Patate" **toright** *"tata"* = *""*
"Patate" **rightof** *"tata"* = *"Patate"*

Groupe 4: Opérateurs additifs.

s + t : concaténation.
s + c **Exemple:** *"Pat"* + *"ate"* = *"Patate"*
c + s

ca **span** *s* : la partie initiale de *s* ne contenant que des caractères du contexte *ca*; si un caractère de *ca* ne débute pas *s*, le résultat est la chaîne vide.
Exemples: *{"Pat"}* **span** *"Patate"* = *"Patat""*
{"ours"} **span** *"Patate"* = *""*
{"Paie"} **span** *"Patate"* = *"Pa""*
{"aie"} **span** *"Patate"* = *""*

ca-s : la chaîne *s* de laquelle on a éliminé toutes les occurences initiales du contexte *ca* ; si *s* ne débute pas par un caractère de *ca*, le résultat est *s*.
Exemples: *{"Pat"}* - *"Patate"* = *"e"*
{"ours"} - *"Patate"* = *"Patate"*
{"Paie"} - *"Patate"* = *"tate"*
{"aie"} - *"Patate"* = *"Patate"*
Remarque: On a un nouvel exemple de coupure.
(ca **span** *s)* + *(ca - s)* = *s*

s - ca : la chaîne *s* de laquelle on a éliminé toutes les occurences finales du contexte *ca*; le résultat est *s* si cette chaîne ne se termine pas par un caractère de *ca*.

s* span *ca : la sous-chaîne de *s* formée de ses occurences finales des caractères de *ca*; le résultat est la chaîne vide si *s* ne se termine par par un tel caractère.
Remarque: On a là un dernier exemple de coupure.
(s - ca) + *(s* **span** *ca)* = *s*
Exemples: *"Patate"* - *{"eti"}* = *"Pata"*
"Patate **span** *{"eti"}* = *"te"*
"Patate" - *{"Pat"}* = *"Patate"*
"Patate" **span** *{"Pat"}* = *""*

Groupe 5. Opérateurs d'optimisation

s* min *t : celle des deux chaînes qui précède l'autre lexicographiquement.
Exemples: *"Patate"* **min** *"Pate"* = *"Patate"*
"Tartine" **min** *"Tarte"* = *"Tarte"*

s* max *t : celle des deux chaînes qui suit l'autre lexicographiquement.
Exemples: *"Patate"* **max** *"Pate"* = *"Pate"*
"Tartine" **max** *"Tarte* = *"Tartine"*

Groupe 6. Comparaisons

s = *t*
s ~ = *t*
s < *t*
s < = *t*
s > *t*
s > = *t*
: ces six opérations sont basées sru le fait que le type *string* est ordonné; une chaîne est considérée comme inférieure à une autre si elle la précède lexicographiquement.
Exemples: *"Patate"* < *"Pate"*
'Tartine"> *"Tarte"*
"Pot" < *"Poteau"*

s* in *a : vrai ssi *s* ne comporte que des caractères de l'alphabet *a*

s* notin *a : vrai ssi *s* contient au moins un caractère qui ne fait pas partie de *a*
Exemples: *"patate"***in** *{***from** *"a"* **to** *"z"}*
"Patate" **notin** *{"a", "t", "e"}*

a* in *s : vrai ssi tous les membres de *a* ont une occurence, au moins, dans la chaîne *s*

a* notin *s : vrai ssi l'alphabet *a* possède un membre (au moins) qui n'apparait pas dans la chaîne *s*.
Exemples: *{"a", "t", "e"}* **in** *"Patate"*
*{***from** *"a"* **to** *"z"}* **notin** *"patate"*
Remarque: Il ressort de ces définitions qu'il ne faut pas utiliser les opérations *a* **in** *s*, *a* **notin** *s* pour savoir si la chaîne *s* possède ou non le contexte *a*; c'est les opérations *a* < > *s* et *a* > < *s* qui remplissent ce rôle.

ct* in *s : vrai ssi *s* contient le contexte *ct*

ct* notin *s : vrai ssi *s* ne contient pas le contexte *ct*

a < > *s*
s < > *a*
: vrai ssi la chaîne *s* contient un membre, au moins de l'alphabet *a*.

a >< *s* : vrai ssi la chaîne *s* ne contient aucun membre de l'alphabet *a*.
s >< *a*

cta **starts** *s* vrai ssi la chaîne s débute par le contexte *cta*.
Exemples: *"P"* **starts** *"Patate"*
~*{"a", "e", "t"}* **starts** *"Patate"*
"" **starts** *"Patate"*
~*{}* **starts** *"Patate"*

cta **ends** *s* : vrai ssi la chaîne *s* finit par le contexte *cta*.
Exemples: *{"a", "e", "t"}* **ends** *"Patate"*
"" **ends** *"Patate"*
"tate" **ends** *"Patate"*
{"e", "f", "g"} **ends** *"Patate"*

Une chaîne peut être convertie, au moyen d'un forceur, dans l'alphabet comportant comme membres les caractères de cette chaîne.

Exemples:

alphabet ["Patate"] = {"P", "a", "e", "t"}
alphabet [""] = {}

On va donner maintenant deux exemples simples de traitement de texte. Le premier de ces programmes *compte_mots* lit un texte sur le fichier de données prédéfini; il comporte, sur chacune de ses lignes, le nombre de mots et il accumule le nombre total de mots du texte. Comme mot, ce programme accepte tout ce qui a la forme d'un identificateur dans les langages de programmation usuels, c'est-à-dire une lettre suivie d'un nombre arbitraire de lettres, de chiffres et/ou de caractères de soulignage.

Les résultats de l'application du programme *compte_mots* à son propre listage sont donnés ici. Chaque ligne débute par le nombre de mots trouvés sur la ligne correspondante du texte donné: ce décompte est suivi d'une flèche à droite de laquelle est reproduite la ligne du texte donné.

Dans le programme lui-même, on remarque les éléments suivants:

- Appliquée à la variable *ligne* de type *string*, la procédure prédéfinie *read* a pour effet de lire une ligne du fichier de données standard et d'en stocker le contenu, sous la forme d'une chaîne, dans cette variable.

- La fonction prédéfinie *end_file* devient vraie lorsque le ficher de données est épuisé.

- La variable *reste* contient la partie de la ligne courante non encore traitée.

- L'assignation *reste* := *reste* **atleft** *lettre* avance jusqu'au début du prochain mot; s'il n'y en a plus, *reste* sera égale à la chaîne vide. On remarque que la convention de coupure choisie lorsque la chaîne considérée *reste* ne contient pas le contexte donné *lettre* n'est révélé commode: elle permet une formulation simple et naturelle de l'algorithme.

- Après avoir compté un mot, on le saute au moyen de l'énoncé *reste* := *alpha_numérique* - *reste*; s'il avait été nécessaire de traiter le mot lui-même, il aurait évidemment été nécessaire de l'extraire au moyen de l'autre partie de la coupure *alpha_ numérique* **span** *reste*.

L'autre programme *parenth* lit également un texte sur le ficher de données standard; il examine si chacune des lignes est parenthésée par rapport aux paires de symboles *()*, *[]*, < > et *{}*. Là aussi, il est fourni le résultat de l'application à son propre listage.

```
***Decompte des mots d'un texte***
 5 mot(s)-->compte_mots                                         Vax Newton
Compiler 0.2                                                    Page    1
 2 mot(s)-->                     Source listing
 0 mot(s)-->
 0 mot(s)-->
 0 mot(s)-->
 5 mot(s)-->      1 /* /*OLDSOURCE=USER2:[RAPIN]COMPTE_MOTS.NEW*/ */
 3 mot(s)-->      1 PROGRAM compte_mots DECLARE
 3 mot(s)-->      4   integer VARIABLE nombre_total:=0,
 1 mot(s)-->     10                    nombre_par_ligne;
10 mot(s)-->     12   CONSTANT lettre={FROM "A" TO "Z", FROM "a" TO "z"},
 5 mot(s)-->     27            alpha_numerique={FROM "A" TO "Z",
 4 mot(s)-->     35                             FROM "a" TO "z",
 0 mot(s)-->     40                             #0123456789#,"_"};
 4 mot(s)-->     45   string VARIABLE ligne,reste
 2 mot(s)-->     50 DO(*compte_mots*)
 8 mot(s)-->     51   print("***Decompte des mots d'un texte***",line);
 3 mot(s)-->     58   UNTIL end_file REPEAT
 2 mot(s)-->     61     read(ligne);
 3 mot(s)-->     66     reste:=ligne; nombre_par_ligne:=0;
 6 mot(s)-->     74     UNTIL (reste:=reste ATLEFT lettre)="" REPEAT
 8 mot(s)-->     85       (*On a trouve le debut d'un mot*)
 3 mot(s)-->     85       nombre_par_ligne:=SUCC nombre_par_ligne;
 3 mot(s)-->     90       (*Enleve le mot*)
 3 mot(s)-->     90       reste:=alpha_numerique-reste
 1 mot(s)-->     95     REPETITION;
 3 mot(s)-->     97     line; edit(nombre_par_ligne,2,0);
 4 mot(s)-->    108            print(_"mot(s)-->",ligne);
 3 mot(s)-->    116     nombre_total:=nombre_total+nombre_par_ligne
 1 mot(s)-->    121   REPETITION;
 3 mot(s)-->    123   line; print("***Total:"_);
 4 mot(s)-->    131         edit(nombre_total,4,0); print(_"mots***")
 2 mot(s)-->    145 DONE(*compte_mots*)
 0 mot(s)-->
 4 mot(s)-->         ****  No messages were issued  ****
***Total:  108 mots***
```

Le gros du travail est réalisé dans la fonction logique *parenthésée*. La chaîne *sujet* contient la partie du texte donné qui doit encore être analysée. L'idée consiste à en éliminer, depuis l'intérieur, les sous-chaînes correctement parenthésées; cette démarche est accomplie, de proche en proche, soit jusqu'à ce que *sujet* devienne vide; soit jusqu'à ce que l'on reconnaisse un défaut de parenthésage.

Initialement, selon la méthode vue dans le programme *compte_mots* , on élimine de *sujet* tous les caractères précédant sa première parenthèse; si *sujet* devient égal à la chaîne vide, l'algorithme est achevé: la chaîne initiale était parenthésée.

Dans le cas contraire, on place dans *début* la sous-chaîne de *sujet* se terminant à sa première parenthèse fermante, puis dans *objet* la sous-chaîne de *début* commençant à sa dernière parenthèse ouvrante; la figure 10 montre quelle serait la situation, à ce moment, si *sujet* était initialement égale à *"x * [a + (b - c) / 2] - z"*.

```
  'TRUE'--->parenth                                            Vax Newton
Compiler 0.2                                                   Page    1
  'TRUE'--->                  Source listing
  'TRUE'--->
  'TRUE'--->
  'TRUE'--->
  'TRUE'--->       1 /* /*OLDSOURCE=USER2:[RAPIN]PARENTH.NEW*/ */
  'TRUE'--->       1 PROGRAM parenth DECLARE
  'TRUE'--->       4
  'TRUE'--->       4   Boolean FUNCTION parenthesee
  'TRUE'--->       7     (string VARIABLE sujet)
 'FALSE'--->      12   (*Retourne TRUE ssi la chaine  sujet  est parenthesse
  'TRUE'--->      12     rapport aux parentheses rondes, aux crochets carres,
  'TRUE'--->      12     aux crochets pointus et aux accolades.
 'FALSE'--->      12   *)
  'TRUE'--->      12   DECLARE(*parenthesee*)
  'TRUE'--->      13     CONSTANT parenthese={"()[]<>{}"},
  'TRUE'--->      20              ouvreur={"([<{"},fermeur={"}>])"},
  'TRUE'--->      32              modele="(),[],<>,{}";
  'TRUE'--->      36     string VARIABLE debut,objet
  'TRUE'--->      41   DO(*parenthesee*)TAKE
  'TRUE'--->      43     CYCLE examen REPEAT
  'TRUE'--->      46       IF
  'TRUE'--->      47         (sujet:=sujet ATLEFT parenthese)=""
  'TRUE'--->      56       EXIT examen TAKE TRUE DONE;
  'TRUE'--->      62
  'TRUE'--->      62       objet:=(debut:=sujet TOLEFT fermeur) ATRIGHT ouvreur;
  'TRUE'--->      74       UNLESS
  'TRUE'--->      75         objet TOLEFT parenthese+objet ATRIGHT parenthese
  'TRUE'--->      82         IN modele
  'TRUE'--->      84       EXIT examen TAKE FALSE DONE;
  'TRUE'--->      90
  'TRUE'--->      90       sujet:=debut RIGHTCUT ouvreur+sujet LEFTCUT fermeur
  'TRUE'--->      99     REPETITION
  'TRUE'--->     100   DONE(*parenthesee*);
  'TRUE'--->     102
  'TRUE'--->     102   string VARIABLE ligne
  'TRUE'--->     105 DO(*parenth*)
  'TRUE'--->     106   UNTIL end_file REPEAT
  'TRUE'--->     109     read(ligne);
 'FALSE'--->     114     print(line,parenthesee(ligne),"--->",ligne)
  'TRUE'--->     127   REPETITION;
  'TRUE'--->     129   print(line,"<<<FIN>>>")
  'TRUE'--->     135 DONE(*parenth*)
  'TRUE'--->
  'TRUE'--->         ****  No messages were issued  ****
<<<FIN>>>
```

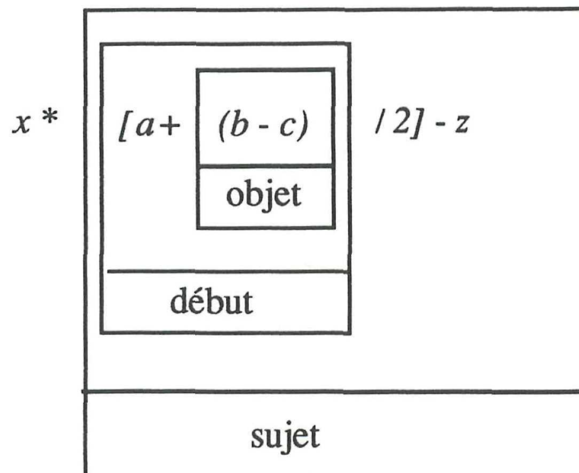


Figure 10

L'expression *objet* **toleft** *parenthèse* + *objet* **atright** *parenthèse* conserve alors la parenthèse initiale et la parenthèse finale de la sous-chaîne *objet*; il suffit ensuite de vérifier si cette concaténation est une sous-chaîne de la chaîne *modèle* = " *{}, [], < >, {}*": si ce n'est pas le cas, la sous chaîne *objet*, et par conséquent la chaîne originale *sujet*, n'est pas parenthésée (on remarque que les virgules, dans le modèle, éviteront d'accepter des couples tels que *)[* à ce stade). Le lecteur est invité à constater que même si la chaîne *sujet* ne contient que des parenthèses ouvrantes, ou que des parenthèses fermantes, ce test décèlera le défaut de parenthésage.

Si la concaténation des parenthèses extrêmes de la chaîne *objet* figure dans *modèle* , il suffit d'éliminer, de la chaîne *sujet*, la sous-chaîne *objet* qui est parenthésée et d'examiner si la chaîne résultante est parenthésée. La nouvelle valeur de *sujet* est obtenue en concaténant la partie initiale non encore traitée *début* **rightcut** *ouvreur* , de la chaîne *début* à la partie finale non encore traitée, *sujet* **leftcut** *fermeur*, de la chaîne *sujet*. On remarque que l'on a utilisé là l'autre partie des coupures *sujet* **toleft** *fermeur* et *début* **atright** *ouvreur* amorcées pour définir les sous-chaînes respectives *début* et *objet*. Dans le cas illustré à la figure 10, il resterait alors dans *sujet* la chaîne "*[a + / 2] - z*" dont il faudrait vérifier le parenthésage.

Chapitre 7

Coroutines

Les coroutines généralisent aussi bien la notion d'objet que celle de procédure. Dans cette dernière optique, une coroutine peut être considérée comme une procédure à points d'arrêt.

Lorsque l'on applique une procédure "normale", celle-ci est entièrement exécutée avant de rendre le contrôle à l'appeleur; pour celà, elle peut, bien entendu, appliquer d'autres procédures voire s'appliquer elle-même récursivement.

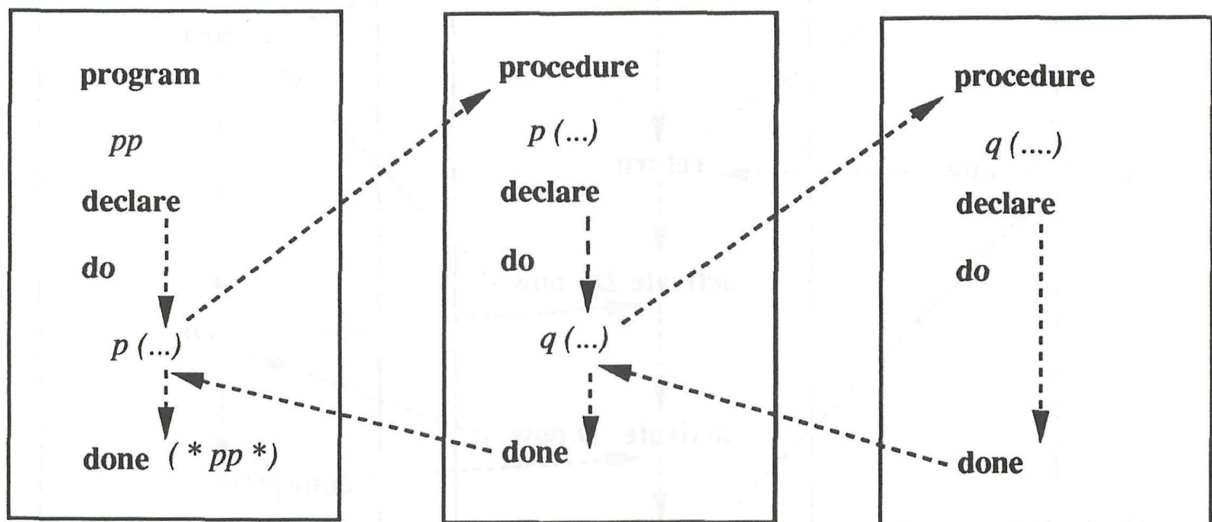


Figure 11

Une coroutine possède en général des points d'arrêts notés au moyen d'une clause **return.** Au point de vue syntaxique, cette clause peut apparaître dans une suite d'énoncés (mais non dans une suite de déclarations), en tout point où un point-virgule est autorisé; un point d'arrêt **return** sépare donc deux énoncés. On notera qu'il est admissible d'incorporer des points d'arrêts dans une procédure si cette dernière est appelée par une coroutine.

Arrivée à un point d'arrêt, une coroutine interrompt son exécution et rend le contrôle au programme qui l'a mise en oeuvre. Au moyen d'un énoncé **activate**, il est possible de reprendre l'exécution d'une coroutine depuis le dernier point d'arrêt où elle s'était interrompue. (Figure 12).

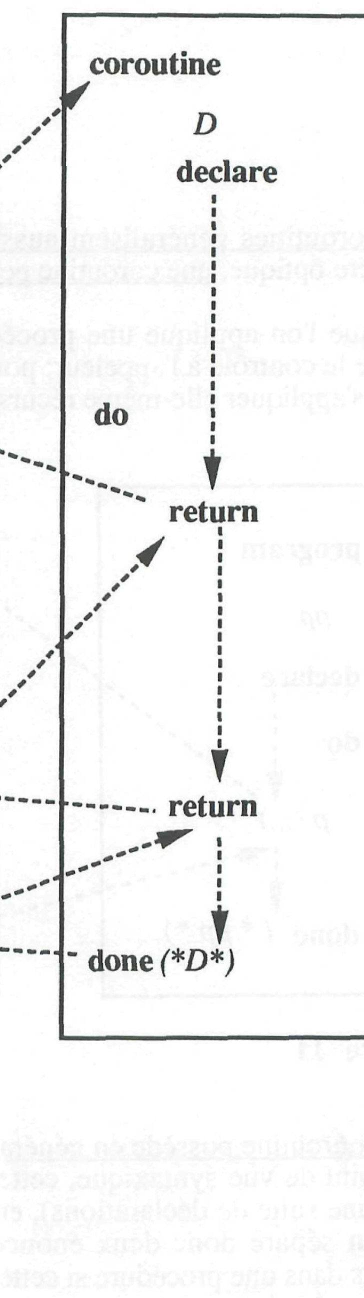


Figure 12

Considéré comme procédure généralisée, une coroutine permet d'établir des structures de contrôle complexes. Lors de l'évolution d'un système, une coroutine peut se trouver dans l'un des états suivants:

- Attachée: c'est l'état d'une coroutine en cours d'exécution; plus précisément, on dira que la coroutine est attachée au programme qui l'a créée ou réactivée au moyen d'un énoncé **activate**. Ainsi, dans le cas de la figure 12, la coroutine *C* est attachée au programme principal *pp* lors de son exécution et la coroutine *D* est attachée à la coroutine *C* lors de la sienne.

- Détachée: c'est l'état d'une coroutine dont l'exécution est temporairement interrompue sur un point d'arrêt **return**. Un énoncé **activate** ne peut porter que sur une coroutine détachée.

- Terminée: c'est l'état d'une coroutine dont l'exécution s'est achevée. Une telle coroutine ne peut plus servir que comme objet passif.

L'état d'une coroutine peut être obtenu en lui applicant l'opérateur **state**; ce résultat a la forme d'une valeur du type scalaire prédéfini *status* suivant:

scalar *status (null, attached, detached, terminated)*

La valeur *null* est l'état de la coroutine vide notée **none**. En-effet, une coroutine est un objet; en particulier, elle peut avoir des attributs. Cet objet appartient à un type processus; sa définition intervient au moyen d'une déclaration de processus de même structure qu'une déclaration de classe: en lieu et place du symbole **class**, une déclaration de processus débute par **process**. Un objet d'un type processus est crée par un générateur de processus; l'effet d'un tel générateur est de créer dynamiquement en mémoire l'objet contenant les entités définies dans la partie formelle et la partie déclarative du type processus, d'initialiser ses paramètres éventuels en les associant aux paramètres effectifs du générateur et d'exécuter l'algorithme inclus dans la déclaration de processus correspondante jusqu'à son prochain point d'arrêt **return** (ou, à défaut d'un tel point) jusqu'à sa fin.

De même qu'une structure de données unique (seule de son type) peut être définie au moyen d'une déclaration de module, il est possible d'utiliser une déclaration de coroutine pour définir une coroutine unique. Une déclaration de coroutine a la même structure qu'une déclaration de module; le symbole **module** y est remplacé par **coroutine**. Une coroutine définie au moyen d'une déclaration de coroutine est automatiquement créée au point de sa déclaration et l'objet correspondant est connecté: comme dans le cas d'un module, ses attributs sont utilisables sans qu'il soit nécessaire de les qualifier par le nom de la coroutine.

Le programme *suite_gen* suivant incorpore une coroutine *suite*; au moyen de l'expression *prochain*, cette coroutine produit les termes successifs de la suite de valeurs entières *2*, *3*, *5*, *7*, *11*, *13*, *17*, *19*, *23*, *25*, ... *6 * k - 1*, *6 * k + 1*, ... Cette coroutine a donc un rôle de générateur de suite.

```
suite_gen                                          Vax Newton Compiler 0.2
Page    1
                         Source listing

 /* /*OLDSOURCE=USER2:[RAPIN]SUITE_GEN.NEW*/ */
 PROGRAM suite_gen DECLARE

 COROUTINE suite
   ATTRIBUTE courant, prochain
 DECLARE(*suite*)
   integer VARIABLE crt VALUE courant:=(RETURN 2);
   (*La valeur du membre courant de la suite; cette valeur n'est
     definie qu'apres la premiere application de  prochain .
   *)

   integer EXPRESSION prochain=
   (*La valeur du prochain element de la suite*)
     (ACTIVATE suite NOW; courant);

   integer VARIABLE k:=0
 DO(*suite*)RETURN
   crt:=3 RETURN
   LOOP k:=SUCC k;
     crt:=6*k-1 RETURN
     crt:=6*k+1 RETURN
   REPETITION
 DONE(*suite*)

 DO(*suite_gen*)
   FOR integer VALUE j FROM 1 TO 100 REPEAT
     edit(prochain,6,0);
     IF j\10=0 THEN line DONE
   REPETITION;
   print("<<<FIN>>>")
 DONE(*suite_gen*)

     ****  No messages were issued  ****
```

Résultats:

```
     2     3     5     7    11    13    17    19    23    25
    29    31    35    37    41    43    47    49    53    55
    59    61    65    67    71    73    77    79    83    85
    89    91    95    97   101   103   107   109   113   115
   119   121   125   127   131   133   137   139   143   145
   149   151   155   157   161   163   167   169   173   175
   179   181   185   187   191   193   197   199   203   205
   209   211   215   217   221   223   227   229   233   235
   239   241   245   247   251   253   257   259   263   265
   269   271   275   277   281   283   287   289   293   295
<<<FIN>>>
```

La coroutine *suite* comporte plusieurs points d'arrêts; le premier intervient à l'intérieur de la partie déclarative, dans la séquence parenthésée responsable d'initialiser à la valeur 2 la variable *crt.* On remarque également la manière dont la coroutine est réactivée, à chaque utilisation de la fonction *prochain*, pour produire l'élément suivant de la suite. On remarque que l'énoncé **activate** a la forme suivante:

activate expression_de_processus **now**

Il est aussi possible d'inclure une suite d'énoncés:

activate
suite_d_énoncés
take expression_de_processus **now**

On remarque également la boucle infinie **loop;** le symbole **loop** peut être considéré comme une abréviation de **while true repeat.**

Les coroutines sont souvent utiles pour simuler un système par l'ordinateur. Le programme *jeu* suivant simule une partie de dés entre un certain nombre de participants. Les règles du jeu sont indiquées, sous la forme d'un commentaire, au début du programme.

```
jeu                                          Vax Newton Compiler 0.2
Page   1
                          Source listing

   1 /* /*OLDSOURCE=USER2:[RAPIN]JEU.NEW*/ */
   1 PROGRAM jeu DECLARE
   4   CONSTANT but=200;
   9 (*Ce programme simule une partie d'un jeu de des; les regles
   9   du jeu sont les suivantes:
   9   -Ce jeu peut etre joue par un nombre arbitraire de joueurs;
   9    ceux_ci jouent a tour de role. Celui qui jouera en premier
   9    est tire au sort.
   9   -Le premier joueur qui a accumule un total d'au moins but
   9    points gagne.
   9   -A son tour de jeu, un joueur doit lancer une ou plusieurs
   9    fois une paire de des.
   9   -Si le joueur obtient un doublet, il accumule aux points
   9    obtenus depuis le debut de ce tour le double des points du
   9    doublet et lance une nouvelle fois les des.
   9   -Cependant, si le joueur obtient trois doublets consecutifs,
   9    il perd les points obtenus depuis le debut de ce tour; de
   9    plus, il doit passer les deux tours de jeu suivants.
   9   -Si le total du dernier lance est egal a 7, ou si le total
   9    accumule depuis le debut du tour de jeu est divisible par
   9    13, le joueur perd les points accumules depuis le debut de
   9    ce tour.
   9   -Si le total du dernier lance est egal a 3, le joueur
   9    accumule a son total les points obtenus depuis le debut du
   9    tour de jeu; il doit passer les 3 tours de jeu suivants.
   9   -Dans tous les autres cas, le joueur accumule aux points
   9    obtenus depuis le debut de ce tour les points du dernier
   9    lancer. Il a ensuite le choix suivant:
   9    1. accumuler a son total les points obtenus depuis le debut
   9       de ce tour.
   9    2. relancer les des
   9 *)
   9   integer EXPRESSION de=1 MAX CEIL(6*random) MIN 6;
  24
  24   CONSTANT nombre_joueurs=8;
  29   Boolean FUNCTOR strategie
  32     (integer VALUE total,points_en_jeu,points_du_lancer);
  42   integer SUBRANGE numero_joueur
  45     (0<=numero_joueur/\numero_joueur<nombre_joueurs);
  55   numero_joueur EXPRESSION un_joueur=
  59     0 MAX FLOOR(nombre_joueurs*random) MIN PRED nombre_joueurs;
  71   /* /*EJECT*/ */
```

```
PROCESS joueur
  ATTRIBUTE nom,total
  (string VALUE nom; strategie VALUE ma_strategie)
DECLARE(*joueur*)
  integer VARIABLE
    tot VALUE total:=0,
    total_en_jeu, points_en_jeu, doublets,
    d_1, d_2, coup;
  PROCEDURE passe_tours(integer VALUE tours)DO
    print(___"il(elle) doit passer"_,edit(tours,1,0),_"tours");
    FOR
      integer FROM 1 TO tours
    REPEAT RETURN
      print(line,nom,_"passe son tour")
    REPETITION
  DONE(*passe_tour*);
  PROCEDURE reste_total DO
    print(line,___"son total reste"_,edit(total,3,0))
  DONE(*reste-total*);
  PROCEDURE accumule_points DO
    print(line,___"son nouveau total est"_,
           edit((tot:=tot+points_en_jeu),3,0))
  DONE(*accumule_points*)
DO(*joueur*)
  CYCLE tour_de_jeu REPEAT
    RETURN(*C'est mon tour de jouer*)
    print(line,nom,_"joue:");
    points_en_jeu:=0;
    CYCLE lancers REPEAT
      doublets:=0;
      CYCLE compte_doubles REPEAT
        print(line,___"il(elle) lance"_,edit((d_1:=de),1,0),
                    _"et"_,edit((d_2:=de),1,0));
      IF d_1~=d_2 EXIT compte_doubles DONE;
      IF (doublets:=SUCC doublets)=3 THEN
        print("; 3e doublet:");
        reste_total; passe_tours(2)
      REPEAT tour_de_jeu DONE;
        print(_"doublet");
        points_en_jeu:=points_en_jeu+4*d_1
      REPETITION(*compte_doubles*);
    IF TAKE
      UNLESS (coup:=d_1+d_2)=7 THEN
        (points_en_jeu:=points_en_jeu+coup)\13=0
      DEFAULT TRUE DONE
    THEN
      reste_total
```

```
jeu                                        Vax Newton Compiler 0.2
Page     3
                   Source listing

        REPEAT tour_de_jeu DONE;

        IF coup=3 THEN
          accumule_points;
          IF
            total>=but
          EXIT tour_de_jeu DONE;

          passe_tours(3)
        REPEAT tour_de_jeu DONE;

          print(";"_,edit(points_en_jeu,3,0),
                _"points en jeu pour le total"_,
                edit((total_en_jeu:=total+points_en_jeu),3,0));
        IF total_en_jeu>=but THEN
          tot:=total_en_jeu
        EXIT tour_de_jeu DONE;

        UNLESS
          ma_strategie[total_en_jeu,points_en_jeu,coup]
        EXIT lancers DONE

        REPETITION(*lancers*);
        accumule_points
      REPETITION(*tour_de_jeu*);
      print(line,___"il(elle) a termine")
    DONE(*joueur*);
    /* /*EJECT*/ */
```

Dans le générateur aléatoire *dé*, on remarque l'expression *1* **max ceil** *(6 * random)* **min** *6;* en principe, l'expression **ceil** *(6 * random)* suffirait: les opérations *1* **max** ... ...**min** *6* garantissent que le résultat restera compris entre *1* et *6* en cas d'un débordement de cet intervalle dû aux imprécisions du calcul en nombre réel du produit *6 * random*.

L'algorithme inclus dans le type processus *joueur* modélise le comportement des joueurs: chaque joueur sera représenté par une instance de ce type. On remarque la manière dont la règle du jeu y est incorporée. Les coroutines du type *joueur* comportent deux points d'arrêts. L'un est au début de la boucle **cycle** *tour_de_jeu*; le joueurs y est arrêté lorsqu'il attend son tour de jeu. Un joueur est arrêté à l'autre clause **return**, dans la procédure *passe_tours*, lorsqu'il doit passer son prochain tour de jeu. Tout en laissant une large place au hasard, ce jeu présente un élément de décision personnelle: on constate, en-effet, que le joueur est amené à choisir s'il veut conserver les points acquis depuis le début du tour de jeu ou s'il préfère continuer à jouer, dans l'espoir d'obtenir un plus grand total, mais avec le risque de perdre les points accumulés depuis le début de ce tour. Il n'est pas évident quelle est la meilleure heuristique; selon son tempérament, chaque joueur adoptera un comportement différent. La stratégie personnelle de chaque joueur est modélisée au moyen d'un objet foncteur *ma_stratégie* du type *stratégie* ; cet objet est communiqué comme paramètre au joueur lors de sa création. Dans ce modèle, on a admis qu'un joueur prenait sa décision en fonction de trois paramètres: le nombre *total* de points accumulés depuis le début de la partie (y compris les points en jeu dans ce tour), le nombre *points_en_jeu* de points acquis depuis le début du tour de jeu et le nombre *coup* de points du dernier jet de dés.

```
   joueur ROW tablee VALUE
     participants=tablee(0 TO PRED nombre_joueurs);
   numero_joueur VARIABLE courant; joueur VARIABLE joueur_courant
 DO(*jeu*)
   (*Place les participants*)
   participants[0]:=
     joueur("Charles",(*La prudence parle*)
            BODY strategie DO TAKE FALSE DONE);
   participants[1]:=
     joueur("Christina",(*Essaye d'evaluer les risques*)
            BODY
              strategie(integer VALUE total,points,coup)
            DO TAKE
              IF points\13=6 THEN
                (*Les deux causes d'ennui seront superposees*)
                TAKE points<50
              DEFAULT(*non, il faut etre plus prudent*)
                TAKE points<30
              DONE
            DONE);
   participants[2]:=
     joueur("Zoom",(*Il faut foncer et gagner*)
            BODY strategie DO TAKE TRUE DONE);
   participants[3]:=
     joueur("Chams",(*Fait une priere et laisse Allah decider*)
            BODY strategie DO TAKE random<5/6 DONE);
   participants[4]:=
     joueur("Samir",(*Prudent au debut; moins a la fin*)
            BODY
              strategie(integer VALUE total,points,coup)
            DO TAKE
              IF total<160 THEN
                TAKE points<20
              DEFAULT TRUE DONE
            DONE);
   participants[5]:=
     joueur("Youcef",(*Demande a Mme. Youcef; comme femme varie... *)
            BODY strategie DO TAKE random>.5 DONE);
   participants[6]:=
     joueur("van Dijk",
            (*Il reve a une belle toile; mais 4 lui porte malheur*)
            BODY
              strategie(integer VALUE total,points,coup)
            DO TAKE coup~=4 DONE);
   participants[7]:=
     joueur("Bertrand",(*Cherche a se mettre en avant*)
            BODY
              strategie(integer VALUE total,points,coup)
            DECLARE
              integer VARIABLE meilleur:=0
            DO
              THROUGH participants VALUE j REPEAT
```

```
jeu                                            Vax Newton Compiler 0.2
Page    5
                    Source listing

  683                 UNLESS j.nom="Bertrand" THEN
  690                   meilleur:=meilleur MAX j.total
  697                 DONE
  698               REPETITION
  699             TAKE
  700               UNLESS total<meilleur THEN
  705                 TAKE points<20
  709               DEFAULT TRUE DONE
  712             DONE);
  715     (*Joue une partie; decide qui joue en premier*)
  715     print(line,"*****Simulation d'une partie de des*****");
  722     randomize;
  724     print(line,"***",
  730           (joueur_courant:= participants[(courant:=un_joueur)]).nom,
  745           _"joue en premier***");
  749     (*Fait avancer le jeu*)
  749     UNTIL
  750       ACTIVATE joueur_courant NOW
  753     TAKE STATE joueur_courant=terminated REPEAT
  759       joueur_courant:=
  761         participants[(courant:=(SUCC courant)\nombre_joueurs)]
  774     REPETITION;
  776     (*Indique le resultat*)
  776     print(line,"***",joueur_courant.nom,_"a gagne***");
  790     THROUGH participants VALUE p REPEAT
  795       print(line,p.nom,":",column(13),edit(p.total,3,0),_"points")
  824     REPETITION;
  826     print(line,"<<<FIN>>>")
  832 DONE(*jeu*)

     ****  No messages were issued  ****
```

La partie exécutable du programme comporte trois phases. Dans la première les joueurs sont crées et stockés dans les composantes successives de la rangée *participants*. En deuxième lieu, la partie est simulée; après avoir choisi aléatoirement le premier joueur, les joueurs sont réactivés à tour de rôle au moyen d'énoncés **activate**: pour cela, on remarque que la variable *courant*, utilisée comme indice dans la rangée *participants* , est incrémentée modulo le nombre de joueurs. Le joueur qui gagne la partie achève son algorithme (les autres ne l'achèveront jamais: on est pas si loin du comportement de joueurs réels!); au moyen d'un test sur l'état du joueur qui vient de jouer, le programme détermine si la partie est achevée. Lorsque c'est le cas, il passe à sa phase finale dans laquelle il imprime le résultat de la partie.

Les résultats d'une partie typique sont donnés ci-après (avec quelques coupures).

```
*****Simulation d'une partie de des*****
***Chams joue en premier***
Chams joue:
   il(elle) lance 3 et 6;   9 points en jeu pour le total   9
   il(elle) lance 1 et 4;  14 points en jeu pour le total  14
   il(elle) lance 6 et 2;  22 points en jeu pour le total  22
   il(elle) lance 6 et 3;  31 points en jeu pour le total  31
   son nouveau total est  31
Samir joue:
   il(elle) lance 3 et 1;   4 points en jeu pour le total   4
   il(elle) lance 6 et 3
   son total reste   0
Youcef joue:
   il(elle) lance 4 et 2;   6 points en jeu pour le total   6
   son nouveau total est   6
van Dijk joue:
   il(elle) lance 1 et 4;   5 points en jeu pour le total   5
   il(elle) lance 6 et 3;  14 points en jeu pour le total  14
   il(elle) lance 2 et 5
   son total reste   0
Bertrand joue:
   il(elle) lance 1 et 6
   son total reste   0
Charles joue:
   il(elle) lance 1 et 5;   6 points en jeu pour le total   6
   son nouveau total est   6
Christina joue:
   il(elle) lance 3 et 3 doublet
   il(elle) lance 2 et 1
   son nouveau total est  15   il(elle) doit passer 3 tours
Zoom joue:
   il(elle) lance 4 et 5;   9 points en jeu pour le total   9
   il(elle) lance 2 et 1
   son nouveau total est  12   il(elle) doit passer 3 tours
Chams joue:
   il(elle) lance 5 et 3;   8 points en jeu pour le total  39
   il(elle) lance 3 et 4
   son total reste  31
Samir joue:
   il(elle) lance 3 et 6;   9 points en jeu pour le total   9
   il(elle) lance 5 et 2
   son total reste   0
Youcef joue:
   il(elle) lance 6 et 1
   son total reste   6
van Dijk joue:
   il(elle) lance 2 et 6;   8 points en jeu pour le total   8
   il(elle) lance 1 et 4
   son total reste   0
Bertrand joue:
   il(elle) lance 3 et 6;   9 points en jeu pour le total   9
   il(elle) lance 2 et 2 doublet
   il(elle) lance 4 et 3
   son total reste   0
Charles joue:
   il(elle) lance 4 et 6;  10 points en jeu pour le total  16
   son nouveau total est  16
Christina passe son tour
Zoom passe son tour
```

```
Chams joue:
   il(elle) lance 4 et 1;   5 points en jeu pour le total  36
   il(elle) lance 2 et 4;  11 points en jeu pour le total  42
   il(elle) lance 1 et 4;  16 points en jeu pour le total  47
   il(elle) lance 6 et 4
   son total reste  31
Samir joue:
   il(elle) lance 5 et 4;   9 points en jeu pour le total   9
   il(elle) lance 5 et 4;  18 points en jeu pour le total  18
   il(elle) lance 1 et 5;  24 points en jeu pour le total  24
   son nouveau total est  24
Youcef joue:
   il(elle) lance 4 et 1;   5 points en jeu pour le total  11
   son nouveau total est  11
van Dijk joue:
   il(elle) lance 2 et 4;   6 points en jeu pour le total   6
   il(elle) lance 4 et 4 doublet
   il(elle) lance 5 et 4;  31 points en jeu pour le total  31
   il(elle) lance 3 et 4
   son total reste   0
Bertrand joue:
   il(elle) lance 2 et 3;   5 points en jeu pour le total   5
   il(elle) lance 2 et 1
   son nouveau total est   8   il(elle) doit passer 3 tours
Charles joue:
   il(elle) lance 4 et 1;   5 points en jeu pour le total  21
   son nouveau total est  21
Christina passe son tour
Zoom passe son tour
Chams joue:
   il(elle) lance 2 et 2 doublet
   il(elle) lance 2 et 3
   son total reste  31
Samir joue:
   il(elle) lance 5 et 2
   son total reste  24
Youcef joue:
   il(elle) lance 1 et 2
   son nouveau total est  14   il(elle) doit passer 3 tours
van Dijk joue:
   il(elle) lance 5 et 5 doublet
   il(elle) lance 3 et 5;  28 points en jeu pour le total  28
   il(elle) lance 5 et 4;  37 points en jeu pour le total  37
   il(elle) lance 2 et 5
   son total reste   0
Bertrand passe son tour
Charles joue:
   il(elle) lance 2 et 4;   6 points en jeu pour le total  27
   son nouveau total est  27
Christina passe son tour
Zoom passe son tour
Chams joue:
   il(elle) lance 5 et 4;   9 points en jeu pour le total  40
   son nouveau total est  40
Samir joue:
   il(elle) lance 1 et 1 doublet
   il(elle) lance 6 et 3
   son total reste  24
Youcef passe son tour
van Dijk joue:
   il(elle) lance 3 et 1;   4 points en jeu pour le total   4
   son nouveau total est   4
Bertrand passe son tour
```

```
Charles joue:
   il(elle) lance 5 et 5 doublet
   il(elle) lance 2 et 2 doublet
   il(elle) lance 4 et 6;  38 points en jeu pour le total  65
   son nouveau total est  65
Christina joue:
   il(elle) lance 4 et 2;   6 points en jeu pour le total  21
   il(elle) lance 3 et 5;  14 points en jeu pour le total  29
   il(elle) lance 4 et 6;  24 points en jeu pour le total  39
   il(elle) lance 3 et 5;  32 points en jeu pour le total  47
   il(elle) lance 6 et 6 doublet
   il(elle) lance 4 et 2;  62 points en jeu pour le total  77
   son nouveau total est  77
Zoom joue:
   il(elle) lance 1 et 2
   son nouveau total est  15   il(elle) doit passer 3 tours

*****E.T.C.*****

Samir joue:
   il(elle) lance 5 et 6;  11 points en jeu pour le total 105
   il(elle) lance 5 et 1;  17 points en jeu pour le total 111
   il(elle) lance 3 et 1;  21 points en jeu pour le total 115
   son nouveau total est 115
Youcef joue:
   il(elle) lance 5 et 5 doublet
   il(elle) lance 3 et 3 doublet
   il(elle) lance 4 et 4; 3e doublet:
   son total reste 119   il(elle) doit passer 2 tours
van Dijk joue:
   il(elle) lance 4 et 1;   5 points en jeu pour le total  44
   il(elle) lance 6 et 4;  15 points en jeu pour le total  54
   il(elle) lance 3 et 6;  24 points en jeu pour le total  63
   il(elle) lance 1 et 2
   son nouveau total est  66   il(elle) doit passer 3 tours
Bertrand joue:
   il(elle) lance 2 et 5
   son total reste   8
Charles joue:
   il(elle) lance 3 et 5;   8 points en jeu pour le total 135
   son nouveau total est 135
Christina passe son tour
Zoom joue:
   il(elle) lance 6 et 4;  10 points en jeu pour le total  28
   il(elle) lance 2 et 2 doublet
   il(elle) lance 6 et 4;  28 points en jeu pour le total  46
   il(elle) lance 2 et 2 doublet
   il(elle) lance 4 et 2;  42 points en jeu pour le total  60
   il(elle) lance 2 et 5
   son total reste  18
Chams passe son tour
Samir joue:
   il(elle) lance 2 et 6;   8 points en jeu pour le total 123
   il(elle) lance 4 et 4 doublet
   il(elle) lance 2 et 4;  30 points en jeu pour le total 145
   son nouveau total est 145
Youcef passe son tour
van Dijk passe son tour

*****E.T.C*****
```

```
Charles joue:
   il(elle) lance 2 et 1
   son nouveau total est 169   il(elle) doit passer 3 tours
Christina joue:
   il(elle) lance 1 et 4;   5 points en jeu pour le total 162
   il(elle) lance 5 et 4;  14 points en jeu pour le total 171
   il(elle) lance 6 et 6 doublet
   il(elle) lance 1 et 1 doublet
   il(elle) lance 3 et 5;  50 points en jeu pour le total 207
   il(elle) a termine
***Christina a gagne***
Charles:     169 points
Christina:   207 points
Zoom:         18 points
Chams:       157 points
Samir:       148 points
Youcef:      151 points
van Dijk:    113 points
Bertrand:     15 points
<<<FIN>>>
```

Pour tirer un enseignement d'une telle simulation, il ne suffit pas de simuler une seule partie. En-effet, en simulant plusieurs parties avec les mêmes personnages, on peut constater que chacun d'entre eux gagne parfois; par contre, il semble que certains joueurs gagnent plus souvent que d'autres. Pour le cerner de plus près, on a transformé le programme *jeu* pour lui faire simuler une suite de *nombre_parties = 1000* parties et de compter le nombre de fois que chacun des joueurs gagne; dans cette version du programme, on a évidemment supprimé l'impression du détail de chaque partie. Ce programme modifié *jeu_stat* est suivi des résultats de trois exécutions distinctes. Ceux-ci montrent, de manière évidente, que certaines stratégies sont préférables à d'autres.

```
jeu_stat                                          Vax Newton Compiler 0.2
Page    1
                         Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]JEU_STAT.NEW*/ */
     1 PROGRAM jeu_stat DECLARE
     4   CONSTANT nombre_parties=1000, but=200;
    13 (*Ce programme simule  nombre_parties  parties d'un jeu de des;
    13   les regles du jeu sont les suivantes:
    13   -Ce jeu peut etre joue par un nombre arbitraire de joueurs;
    13    ceux_ci jouent a tour de role. Celui qui jouera en premier
    13    est tire au sort.
    13   -Le premier joueur qui a accumule un total d'au moins  but
    13    points gagne.
    13   -A son tour de jeu, un joueur doit lancer une ou plusieurs
    13    fois une paire de des.
    13   -Si le joueur obtient un doublet, il accumule aux points
    13    obtenus depuis le debut de ce tour le double des points du
    13    doublet et lance une nouvelle fois les des.
    13   -Cependant, si le joueur obtient trois doublets consecutifs,
    13    il perd les points obtenus depuis le debut de ce tour; de
    13    plus, il doit passer les deux tours de jeu suivants.
    13   -Si le total du dernier lance est egal a 7, ou si le total
    13    accumule depuis le debut du tour de jeu est divisible par
    13    13, le joueur perd les points accumules depuis le debut de
    13    ce tour.
    13   -Si le total du dernier lance est egal a 3, le joueur
    13    accumule a son total les points obtenus depuis le debut du
    13    tour de jeu; il doit passer les 3 tours de jeu suivants.
    13   -Dans tous les autres cas, le joueur accumule aux points
    13    obtenus depuis le debut de ce tour les points du dernier
    13    lancer. Il a ensuite le choix suivant:
    13    1. accumuler a son total les points obtenus depuis le debut
    13       de ce tour.
    13    2. relancer les des
    13 *)
    13   integer EXPRESSION de=1 MAX CEIL(6*random) MIN 6;
    28
    28   CONSTANT nombre_joueurs=8;
    33   Boolean FUNCTOR strategie
    36     (integer VALUE total,points_en_jeu,points_du_lancer);
    46   integer SUBRANGE numero_joueur
    49     (0<=numero_joueur/\numero_joueur<nombre_joueurs);
    59   numero_joueur EXPRESSION un_joueur=
    63     0 MAX FLOOR(nombre_joueurs*random) MIN PRED nombre_joueurs;
    75   /* /*EJECT*/ */
```

En particulier, il faut être ni trop timide, ni trop téméraire. Des stratégies proposées, la meilleure est clairement celle de *Christina*; les seuls autres qui mériteraient d'être affinées sont celles de *Bertrand* et de *Samir*. On pourrait conseiller à *Bertrand* de ne pas chercher à rattraper son retard en un seul tour de jeu si ce retard est trop grand.

```
jeu_stat                                    Vax Newton Compiler 0.2
Page    2
                  Source listing

    PROCESS joueur
      ATTRIBUTE nom,total
      (string VALUE nom; strategie VALUE ma_strategie)
    DECLARE(*joueur*)
      integer VARIABLE
        tot VALUE total:=0,
        total_en_jeu, points_en_jeu, doublets,
        d_1, d_2, coup;

      PROCEDURE passe_tours(integer VALUE tours)DO
        FOR
          integer FROM 1 TO tours
        REPEAT RETURN REPETITION
      DONE(*passe_tour*)
    DO(*joueur*)
      CYCLE tour_de_jeu REPEAT
        RETURN(*C'est mon tour de jouer*)
        points_en_jeu:=0;
        CYCLE lancers REPEAT
          doublets:=0;
          CYCLE compte_doubles REPEAT
          IF (d_1:=de)~=(d_2:=de) EXIT compte_doubles DONE;

          IF (doublets:=SUCC doublets)=3 THEN
            passe_tours(2)
          REPEAT tour_de_jeu DONE;

            points_en_jeu:=points_en_jeu+4*d_1
          REPETITION(*compte_doubles*);
        IF TAKE
          UNLESS (coup:=d_1+d_2)=7 THEN
            (points_en_jeu:=points_en_jeu+coup)\13=0
          DEFAULT TRUE DONE
        REPEAT tour_de_jeu DONE;

        IF coup=3 THEN
          tot:=tot+points_en_jeu;
          IF
            total>=but
          EXIT tour_de_jeu DONE;

          passe_tours(3)
        REPEAT tour_de_jeu DONE;

        IF (total_en_jeu:=total+points_en_jeu)>=but THEN
          tot:=total_en_jeu
        EXIT tour_de_jeu DONE;

        UNLESS
          ma_strategie[total_en_jeu,points_en_jeu,coup]
        EXIT lancers DONE
```

```
jeu_stat                                          Vax Newton Compiler 0.2
Page    3
                        Source listing

          REPETITION(*lancers*);
          tot:=tot+points_en_jeu
        REPETITION(*tour_de_jeu*)
      DONE(*joueur*);
      /* /*EJECT*/ */
```

D'autre part, on pourrait chercher à optimiser les valeurs des paramètres numériques intervenant dans les trois stratégies retenues; on pourrait chercher à combiner les aspects positifs de ces trois stratégies en une stratégie plus complexe.

On remarque aussi que la valeur de la stratégie adoptée par un joueur n'est pas toujours réflétée par le total du nombre de points accumulés en un grand nombre de parties. Ainsi, avec sa stratégie trop prudente, *Charles* accumule plus de points que *Zoom* avec sa stratégie trop impétueuse; il gagne cependant moins souvent que lui. De-même, *Bertrand* gagne beaucoup plus souvent que ne le laisse penser le nombre de points relativement faible qu'il accumule; sa stratégie est, en-effet, une stratégie à risque: s'il arrive à prendre de l'avance sur les autres joueurs en début de partie, elle se révélera excellente; par contre, s'il prend du retard en début de partie, il va tout risquer pour le rattraper en un seul tour de jeu aboutissant à un comportement impétueux analogue à celui de *Zoom* (c'est ce qui a dû arriver dans la partie dont de larges extraits ont été donnés et où *Bertrand* obtient un score misérable).

```
jeu_stat                                    Vax Newton Compiler 0.2
Page    4
                   Source listing

   joueur ROW tablee VALUE
     participants=tablee(0 TO PRED nombre_joueurs);
   numero_joueur VARIABLE courant; joueur VARIABLE joueur_courant;

   integer ROW compteurs VALUE
     nombre_gains=THROUGH
                     compteurs(0 TO PRED nombre_joueurs):=0
                   REPETITION,
     nombre_points=THROUGH
                      compteurs(0 TO PRED nombre_joueurs):=0
                    REPETITION
 DO(*jeu_stat*) randomize;
   (*Simule l'ensemble des parties*)
   print("*****Simule"_,edit(nombre_parties,4,0),
         _"parties d'un jeu de des*****");
   FOR integer FROM 1 TO nombre_parties REPEAT
     (*Place les participants*)
     participants[0]:=
       joueur("Charles",(*La prudence parle*)
              BODY strategie DO TAKE FALSE DONE);
     participants[1]:=
       joueur("Christina",(*Essaye d'evaluer les risques*)
              BODY
                strategie(integer VALUE total,points,coup)
              DO TAKE
                IF points\13=6 THEN
                  (*Les deux causes d'ennui seront superposees*)
                  TAKE points<50
                DEFAULT(*non, il faut etre plus prudent*)
                  TAKE points<30
                DONE
              DONE);
     participants[2]:=
       joueur("Zoom",(*Il faut foncer et gagner*)
              BODY strategie DO TAKE TRUE DONE);
     participants[3]:=
       joueur("Chams",(*Fait une priere et laisse Allah decider*)
              BODY strategie DO TAKE random<5/6 DONE);
     participants[4]:=
       joueur("Samir",(*Prudent au debut; moins a la fin*)
              BODY
                strategie(integer VALUE total,points,coup)
              DO TAKE
                IF total<160 THEN
                  TAKE points<20
                DEFAULT TRUE DONE
              DONE);
     participants[5]:=
       joueur("Youcef",(*Demande a Mme. Youcef; comme femme varie... *)
              BODY strategie DO TAKE random>.5 DONE);
     participants[6]:=
       joueur("van Dijk",
```

```
jeu_stat                                          Vax Newton Compiler 0.2
Page    5
                    Source listing

   533                (*Il reve a une belle toile; mais 4 lui porte malheur*)
   533                BODY
   534                  strategie(integer VALUE total,points,coup)
   544                DO TAKE coup~=4 DONE);
   552       participants[7]:=
   557         joueur("Bertrand",(*Cherche a se mettre en avant*)
   561                BODY
   562                  strategie(integer VALUE total,points,coup)
   572                DECLARE
   573                  integer VARIABLE meilleur:=0
   578                DO
   579                  THROUGH participants VALUE j REPEAT
   584                    UNLESS j.nom="Bertrand" THEN
   591                      meilleur:=meilleur MAX j.total
   598                    DONE
   599                  REPETITION
   600                TAKE
   601                  UNLESS total<meilleur THEN
   606                    TAKE points<20
   610                  DEFAULT TRUE DONE
   613                DONE);
   616       (*Joue une partie; decide qui joue en premier*)
   616       joueur_courant:=participants[(courant:=un_joueur)];
   627       (*Fait avancer le jeu*)
   627       UNTIL
   628         ACTIVATE joueur_courant NOW
   631       TAKE STATE joueur_courant=terminated REPEAT
   637         joueur_courant:=
   639           participants[(courant:=(SUCC courant)\nombre_joueurs)]
   652       REPETITION;
   654       (*Indique le resultat*)
   654       nombre_gains[courant]:=SUCC nombre_gains[courant];
   665       THROUGH participants INDEX courant VALUE joueur_courant REPEAT
   672         nombre_points[courant]:=
   677           nombre_points[courant]+joueur_courant.total
   685       REPETITION
   686     REPETITION;
   688     (*Imprime les statistiques*)
   688     print(line,line,"Nom",column(13),"Gains",column(25),"Points",line);
   713     THROUGH participants INDEX courant VALUE joueur_courant REPEAT
   720       print(line,joueur_courant.nom,
   728             column(14),edit(nombre_gains[courant],4,0),
   745             column(25),edit(nombre_points[courant],6,0))
   762     REPETITION;
   764     print(line,"<<<FIN>>>")
   770   DONE(*jeu_stat*)

      ****  No messages were issued  ****
```

```
*****Simule 1000 parties d'un jeu de des*****
Nom           Gains       Points

Charles          18        96028
Christina       236       129771
Zoom             47        54482
Chams           127        98730
Samir           203       129107
Youcef           93       108395
van Dijk         70        80766
Bertrand        206       106190
<<<FIN>>>

*****Simule 1000 parties d'un jeu de des*****
Nom           Gains       Points

Charles          20        98027
Christina       248       131997
Zoom             43        54306
Chams           117       101363
Samir           211       129469
Youcef          100       111006
van Dijk         73        81044
Bertrand        188       101026
<<<FIN>>>

*****Simule 1000 parties d'un jeu de des*****
Nom           Gains       Points

Charles          23        96050
Christina       223       133281
Zoom             53        56844
Chams           101        97284
Samir           195       130896
Youcef          111       112128
van Dijk         72        78809
Bertrand        222       107203
<<<FIN>>>
```

Dans le programme *gen_mots* suivant intervient un type processus récursif *génère_mots*. On considère un alphabet *alpha*; on veut réaliser une coroutine qui génère successivement tous les mots construits: à partir des caractères de cet alphabet. La première chose à décider est l'ordre dans lequel les mots seront produits; sauf si l'alphabet ne comporte qu'un seul caractère (ou s'il est vide!), il n'est évidemment pas possible de les produire dans l'ordre lexicographique croissant. On peut par contre les produire dans l'ordre croissant de leurs longueur; pour chaque longueur, les mots seront produits dans l'ordre lexicographique croissant. C'est cet ordre qui a été programmé dans le type processus *génère_mots*; lors de la création d'un objet de ce type, il lui est fourni comme paramètre l'alphabet considéré. Après sa création, une coroutine du type *génère_mots* se détache sur le point d'arrêt placé immédiatement avant l'initialisation à la chaîne vide de la variable *crt*. La coroutine est ensuite réactivée par l'intermédiaire de sa fonction attribut *prochain*; on voit que l'on a utilisé là le même truc que dans le programme *suite_gen*. Au premier usage de cette fonction attribut, il sera livré la chaîne vide; la coroutine sera alors détachée sur le point d'arrêt qui précède la définition de la valeur *fils*. Cette dernière valeur est une nouvelle coroutine du même type et chargée de produire les mêmes mots. Le partage des tâches entre une coroutine du type *génère_mots* et le *fils* qu'elle a engendré est le suivant: la

première produit directement le dernier caractère du mot qu'elle doit générer; la coroutine *fils* est chargée de produire tout ce qui précède le dernier caractère. Ceci est mis en place dans la partie exécutable des processus *génère_mots*; celle-ci comporte une boucle infinie. Lors d'une itération de cette boucle, il s'agit de produire tous les mots obtenus à partir du même mot du processus *fils*.

```
gen_mots                                          Vax Newton Compiler 0.2
Page    1
                  Source listing

/* /*OLDSOURCE=USER2:[RAPIN]GEN_MOTS.NEW*/ */
PROGRAM gen_mots DECLARE

  PROCESS genere_mots VALUE moi
    ATTRIBUTE courant,prochain
    (alphabet VALUE alpha)
  (*Genere l'ensemble des mots construits au moyen de l'alphabet
    alpha  : les mots sont produits dans l'ordre des longueurs
    croissantes, a commencer par le mot vide; les mots de meme
    longueur sont produits dans l'ordre lexicographique croissant.
  *)
  DECLARE(*genere_mots*)
    string VARIABLE mot_crt VALUE courant:=(RETURN "");
    (*Le mot produit en dernier lieu par le generateur;  courant
      n'est pas defini avant la premiere application de  prochain .
    *)

    string EXPRESSION prochain=
    (*Obtient le prochain mot du generateur*)
      (ACTIVATE moi NOW; courant);

    genere_mots VALUE fils=(RETURN genere_mots(alpha))
  DO(*genere_mots*)LOOP
    fils.prochain;
    THROUGH alpha VALUE car REPEAT
      mot_crt:=fils.courant+car
    RETURN REPETITION
  REPETITION(*genere_mots*)DONE VALUE
    gen_abc=genere_mots({"a","b","c"}),
    gen_recto=genere_mots({"recto"})
DO(*gen_mots*)
  print(#***Mots de l'alphabet {"a","b","c"}***#);
  FOR integer VALUE k FROM 0 TO 99 REPEAT
    IF k\10=0 THEN line DEFAULT column(5*(k\10)+1) DONE;
    print(gen_abc.prochain)
  REPETITION;
  print(line,#***Mots de l'alphabet {"recto"}***#);
  FOR integer VALUE k FROM 0 TO 399 REPEAT
    IF k\10=0 THEN line DEFAULT column(5*(k\10)+1) DONE;
    print(gen_recto.prochain)
  REPETITION;
  print(line,"<<<FIN>>>")
DONE(*gen_mots*)

  ****  No messages were issued  ****
```

Résultats:

```
***Mots de l'alphabet {"a","b","c"}***
     a    b    c    aa   ab   ac   ba   bb   bc
ca   cb   cc   aaa  aab  aac  aba  abb  abc  aca
acb  acc  baa  bab  bac  bba  bbb  bbc  bca  bcb
bcc  caa  cab  cac  cba  cbb  cbc  cca  ccb  ccc
aaaa aaab aaac aaba aabb aabc aaca aacb aacc abaa
abab abac abba abbb abbc abca abcb abcc acaa acab
acac acba acbb acbc acca accb accc baaa baab baac
baba babb babc baca bacb bacc bbaa bbab bbac bbba
bbbb bbbc bbca bbcb bbcc bcaa bcab bcac bcba bcbb
bcbc bcca bccb bccc caaa caab caac caba cabb cabc
***Mots de l'alphabet {"recto"}***
     c    e    o    r    t    cc   ce   co   cr
ct   ec   ee   eo   er   et   oc   oe   oo   or
ot   rc   re   ro   rr   rt   tc   te   to   tr
tt   ccc  cce  cco  ccr  cct  cec  cee  ceo  cer
cet  coc  coe  coo  cor  cot  crc  cre  cro  crr
crt  ctc  cte  cto  ctr  ctt  ecc  ece  eco  ecr
ect  eec  eee  eeo  eer  eet  eoc  eoe  eoo  eor
eot  erc  ere  ero  err  ert  etc  ete  eto  etr
ett  occ  oce  oco  ocr  oct  oec  oee  oeo  oer
oet  ooc  ooe  ooo  oor  oot  orc  ore  oro  orr
ort  otc  ote  oto  otr  ott  rcc  rce  rco  rcr
rct  rec  ree  reo  rer  ret  roc  roe  roo  ror
rot  rrc  rre  rro  rrr  rrt  rtc  rte  rto  rtr
rtt  tcc  tce  tco  tcr  tct  tec  tee  teo  ter
tet  toc  toe  too  tor  tot  trc  tre  tro  trr
trt  ttc  tte  tto  ttr  ttt  cccc ccce ccco cccr
ccct ccec ccee cceo ccer ccet ccoc ccoe ccoo ccor
ccot ccrc ccre ccro ccrr ccrt cctc ccte ccto cctr
cctt cecc cece ceco cecr cect ceec ceee ceeo ceer
ceet ceoc ceoe ceoo ceor ceot cerc cere cero cerr
cert cetc cete ceto cetr cett cocc coce coco cocr
coct coec coee coeo coer coet cooc cooe cooo coor
coot corc core coro corr cort cotc cote coto cotr
cott crcc crce crco crcr crct crec cree creo crer
cret croc croe croo cror crot crrc crre crro crrr
crrt crtc crte crto crtr crtt ctcc ctce ctco ctcr
ctct ctec ctee cteo cter ctet ctoc ctoe ctoo ctor
ctot ctrc ctre ctro ctrr ctrt cttc ctte ctto cttr
cttt eccc ecce ecco eccr ecct ecec ecee eceo ecer
ecet ecoc ecoe ecoo ecor ecot ecrc ecre ecro ecrr
ecrt ectc ecte ecto ectr ectt eecc eece eeco eecr
eect eeec eeee eeeo eeer eeet eeoc eeoe eeoo eeor
eeot eerc eere eero eerr eert eetc eete eeto eetr
eett eocc eoce eoco eocr eoct eoec eoee eoeo eoer
eoet eooc eooe eooo eoor eoot eorc eore eoro eorr
eort eotc eote eoto eotr eott ercc erce erco ercr
erct erec eree ereo erer eret eroc eroe eroo eror
erot errc erre erro errr errt ertc erte erto ertr
ertt etcc etce etco etcr etct etec etee eteo eter
etet etoc etoe etoo etor etot etrc etre etro etrr
<<<FIN>>>
```

Pour cela, on commence par produire le prochain mot du *fils*: à ce sujet, il aurait été possible de remplacer l'énoncé *fils. prochain* par **activate** *fils* **now** puisque la valeur produite par le générateur *fils* n'est pas immédiatement utilisée en ce point. Le parcours subséquent, au moyen d'un itérateur **through**, de l'alphabet donné *alpha* permet d'obtenir les caractères qui doivent

successivement être appondondus au mot *fils.courant*. Bien entendu, un point d'arrêt **return** suit immédiatement l'assignation d'une nouvelle valeur à la variable *crt*. On peut remarquer qu'un processus *génère_mots* et le processus *fils* qu'il a engendré seront chaînés en une liste; en pratique, cette liste ne sera jamais très longue: elle comporte un membre pour chaque caractère du mot produit par le processus considéré plus un membre pour le mot vide initial. On se rend facilement compte que, sauf dans le cas d'un alphabet possédant un seul caractère, un volume de calcul considérable est nécessaire pour l'obtention de mots (modérément) longs au moyen d'un tel générateur.

Chapitre 8

Tables associatives

Une table est un objet analogue à une rangée. De même qu'une rangée, une table est un ensemble de variables du même type de base. Il y a cependant deux différences essentielles entre les tables et les rangées. Tout d'abord, tandis que les composantes d'une rangée sont indicées par des valeurs entières les éléments d'une table sont indicés au moyen de chaînes de caractères. D'autre part, au moment de la création d'une table, il n'est spécifié que la capacité de cette dernière; les chaînes susceptibles d'indicer les composantes individuelles d'une table ne sont enregistrées qu'à la première utilisation de la variable indicées correspondante.

Exemple:

real **table** *real_table* **value** *rt* = *real_table (10)*

Cette déclaration définit le type *real_table ;* les objets de ce type sont des tables de variables réelles. De plus, il est déclaré une table spécifique *rt* d'une capacité de dix composantes réelles. Un type table est déclaré comme suit:

indication_de_type **table** identificateur

Une table spécifique est créée au moyen d'un générateur de table de la forme:

type_table *(*expression_entière*)*

Les opérations suivantes sont disponibles sur les tables prédéfinies du langage Newton.
Dans la description des opérations subséquentes on supposera que *t* est une table et *s* une chaîne de caractères:

Interrogateurs:

empty *t* - Vrai ssi la table *t* est vide; immédiatement après sa création, une table est nécessairement vide.

full *t* - Vrai ssi la table *t* est pleine; dans cet état, il n'est plus possible d'y placer de nouvelles composantes.

card *t* - Le nombre de composantes de la table *t*.

capacity *t* - La capacité maximum de la table *t* .

t **entry** *s* - Vrai ssi la table *t* possède une composante indicée par la chaîne *s*; immédiatement après la création d'une table *t*, l'expression *t* **entry** *s* sera fausse quelle que soit la chaîne *s*.

Remarque:

On a les équivalences suivantes:

empty *t* = = **card** *t* = *0*
full *t* = = **card***t* = **capacity** *t*

Sélecteur:

t [s] - Résulte dans l'élément de la table *t* indicé par la chaîne *s* . Si la condition *t* **entry** *s* est initialement fausse, l'expression *t[s]* joue tout d'abord le rôle de constructeur; une variable appropriée est insérée dans la table. Il y aura erreur à l'exécution ssi **full** *t* est initialement vraie et si, de plus, *t* **entry** *s* est fausse.

Itérateur:

```
through t
   index s
   value ts
   reference rts
   := expression
repeat
   suite_d_énoncés
repetition
```

- Cet itérateur prédéfini présente les mêmes possibilités que celui sur les rangées. La suite d'énoncés entre **repeat** et **repetition** est exécutée une fois pour chacune des composantes de la table concernée *t* ; dans cette suite, l'identificateur *s* introduit par la clause **index** a pour valeur la chaîne égale à l'indice de la composante courante et l'identificateur *ts* dans la clause **value** y représente la valeur stockée dans cette dernière. L'identificateur *rts dans la clause* **reference** est un repère à la composante courante; l'expression après le := est évaluée: la valeur résultante est stockée dans la composante courante de la table. Les clauses **index** *s*, **value** *ts*, **reference** *rts* et := expression sont toutes facultatives.

Remarque:

Lors d'une itération, au moyen d'un énoncé **through**, sur les éléments d'une table, l'ordre de prise en compte de ces derniers n'est pas défini à priori. En particulier, il n'y a aucune raison de supposer qu'ils seront traités dans l'ordre lexicographique croissant des chaînes servant d'indice ou dans l'ordre de leur insertion dans la table.

Exemple

On considère la table *rt* d'une capacité de dix composantes réelles définie dans l'exemple précédent; on suppose que l'on exécute la séquence suivante:

```
rt ["Toto"] := 5.2; rt ["Zut"];
rt ["Titi"] := 2 * rt ["Toto"];
rt ["Tintin"] := 7.1;
rt ["Titi"] := rt ["Titi"] - rt ["Tintin"]
```

A ce stade, on aura la situation représentée à la figure 13. Dans cette figure, on peut relever que l'entité *rt ["Zut"]* a été utilisée comme constructeur pur; une variable indicée par la chaîne *"Zut"* a été insérée dans la table, mais aucune valeur n'a été stockée dans cette variable. A ce stade, les relations suivantes seront toutes vraies:

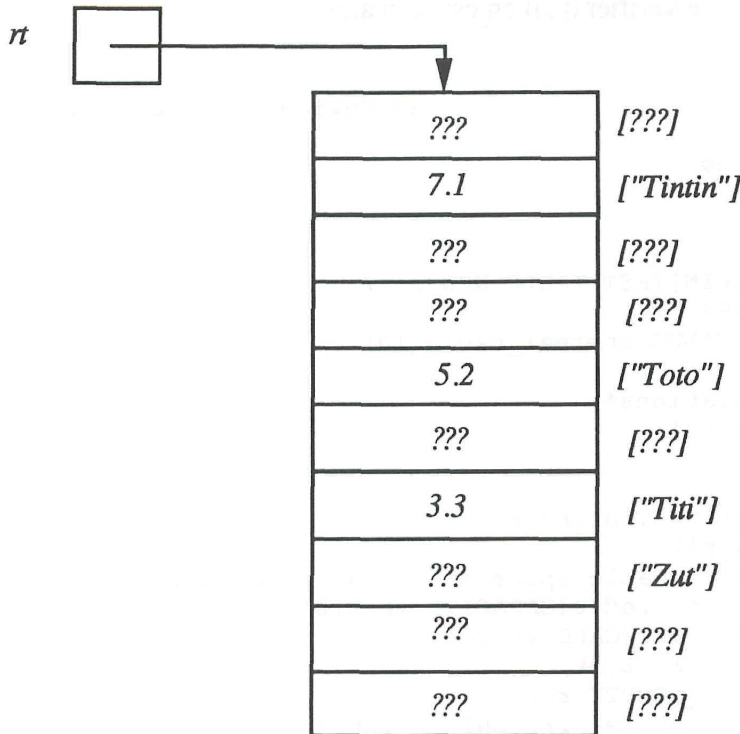


Figure 13

capacity *rt = 10*
card *rt = 4*
~ **full** *rt*
~ **empty** *rt*
rt **entry** *"Toto"*
rt ["Toto"] = 5.2
rt **entry** *"Zut"*
rt **entry** *"Titi"*
rt ["Titi"] = 3.3
rt **entry** *"Tintin"*
rt [Tintin"] = 7.1

A ce stade, la valeur de l'expression *rt ["Zut"]* n'est pas définie; si l'on fait , de plus, l'assignation *rt ["Zut] := rt ["Toto"] - rt ["Tintin"]* l'on obtiendra *rt ["Zut"] = - 1.9*. On suppose que, dans cette nouvelle situation, on effectue l'itération suivante:

```
through
   rt index s value rts
repeat
   print (line, s, column (10), edit (rts, 5, 1))
repetition
```

Il sera imprimé, dans un ordre non fixé à priori, les quatre lignes de résultats suivantes:

```
Tintin     7.1
Toto       5.2
Titi       3.3
Zut       -1.9
```

Le programme *test-table* suivant permet de vérifier qu'il en est bien ainsi.

```
test_table                                          Vax Newton Compiler 0.2
Page    1
                   Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]TEST_TABLE.NEW*/ */
     1 PROGRAM test_table DECLARE
     4   real TABLE real_table VALUE rt=real_table(10)
    14 DO(*test_table*)
    15   (*Fait quelques manipulations*)
    15   rt["Toto"]:=5.2; rt["Zut"];
    29   rt["Titi"]:=2*rt["Toto"];
    41   rt["Tintin"]:=7.1;
    50   rt["Titi"]:=rt["Titi"]-rt["Tintin"];
    65   (*Imprime l'etat courant*)
    65   print(line,"***Etat de la table apres quelques manipulations***",
    71         line,"CAPACITY rt ="_,edit(CAPACITY rt,3,0),
    86         line,"CARD rt ="_,edit(CARD rt,3,0),
   101         line,"FULL rt =="_,FULL rt,
   109         line,"EMPTY rt =="_,EMPTY rt,
   117         line,#rt ENTRY "Toto" ==#_,rt ENTRY "Toto",
   126         line,#rt["Toto"] =#_,edit(rt["Toto"],5,1),
   143         line,#rt ENTRY "Zut" ==#_,rt ENTRY "Zut",
   152         line,#rt ENTRY "QQ" ==#_,rt ENTRY "QQ",
   161         line,#rt ENTRY "Titi" ==#_,rt ENTRY "Titi",
   170         line,#rt["Titi"] =#_,edit(rt["Titi"],5,1),
   187         line,#rt ENTRY "Tintin" ==#_,rt ENTRY "Tintin",
   196         line,#rt["Tintin"] =#_,edit(rt["Tintin"],5,1));
   214   (*Met quelque chose dans  rt["Zut"] *)
   214   rt["Zut"]:=rt["Toto"]-rt["Tintin"];
   229   (*Imprime le contenu des elements*)
   229   print(line,"***Contenu final des elements***");
   236   THROUGH
   237     rt INDEX s VALUE rts
   242   REPEAT
   243     print(line,s,column(10),edit(rts,5,1))
   263   REPETITION;
   265   print(line,"<<<FIN>>>")
   271 DONE(*test_table*)

    **** No messages were issued ****
```

Résultats:

```
***Etat de la table apres quelques manipulations***
CAPACITY rt =  10
CARD rt =    4
FULL rt ==  'FALSE'
EMPTY rt ==  'FALSE'
rt ENTRY "Toto" ==   'TRUE'
rt["Toto"] =   5.2
rt ENTRY "Zut" ==   'TRUE'
rt ENTRY "QQ" ==  'FALSE'
rt ENTRY "Titi" ==   'TRUE'
rt["Titi"] =  3.3
rt ENTRY "Tintin" ==   'TRUE'
rt["Tintin"] =   7.1
***Contenu final des elements***
Titi        3.3
Tintin      7.1
Toto        5.2
Zut        -1.9
<<<FIN>>>
```

On peut se demander comment implanter en pratique un type table; il est en particulier souhaitable que l'accès aux composantes individuelles soit efficace. Une technique d'implantation possible est basée sur les fonctions de hachage. Soit ***capacité*** le nombre maximum d'éléments qu'il est prévu d'insérer dans la table; les éléments seront stockés dans une rangée ***éléments*** , indicée par des valeurs entières incluses entre *1* et ***capacité***, ou mieux entre *0* et *capacité -1*. On définit une fonction *hache* qui, pour chaque chaîne susceptible d'être utilisée comme indice d'un élément de la table, livre une valeur entière comprise entre *0* et *capacité -1*; l'idée consiste évidemment à utiliser le résultat de cette fonction pour indicer la rangée représentative *éléments*. A priori, il est soutraitable que la fonction de hachage *hache* satisfaisse aux conditions suivantes:

- Rapide à évaluer
- Donne des résultat différents pour toute paire de chaîne utilisée comme indices dans la table.

Si la première condition n'est déjà pas facile à satisfaire en pratique, la seconde ne peut être garantie que si l'on connaît à priori le sous-ensemble des chaînes qui seront effectivement utilisées comme indices dans la table. Comme cette dernière contrainte est trop restrictive, on est obligé d'admettre la possibilité que la fonction de hachage applique certaines paires de chaînes utilisées pour indicer la table sur la même valeur entière; les composantes correspondantes de la table seraient alors implantées au moyen du même élément de la rangée représentative: on dira que de telles composantes sont en collision (figure 14).

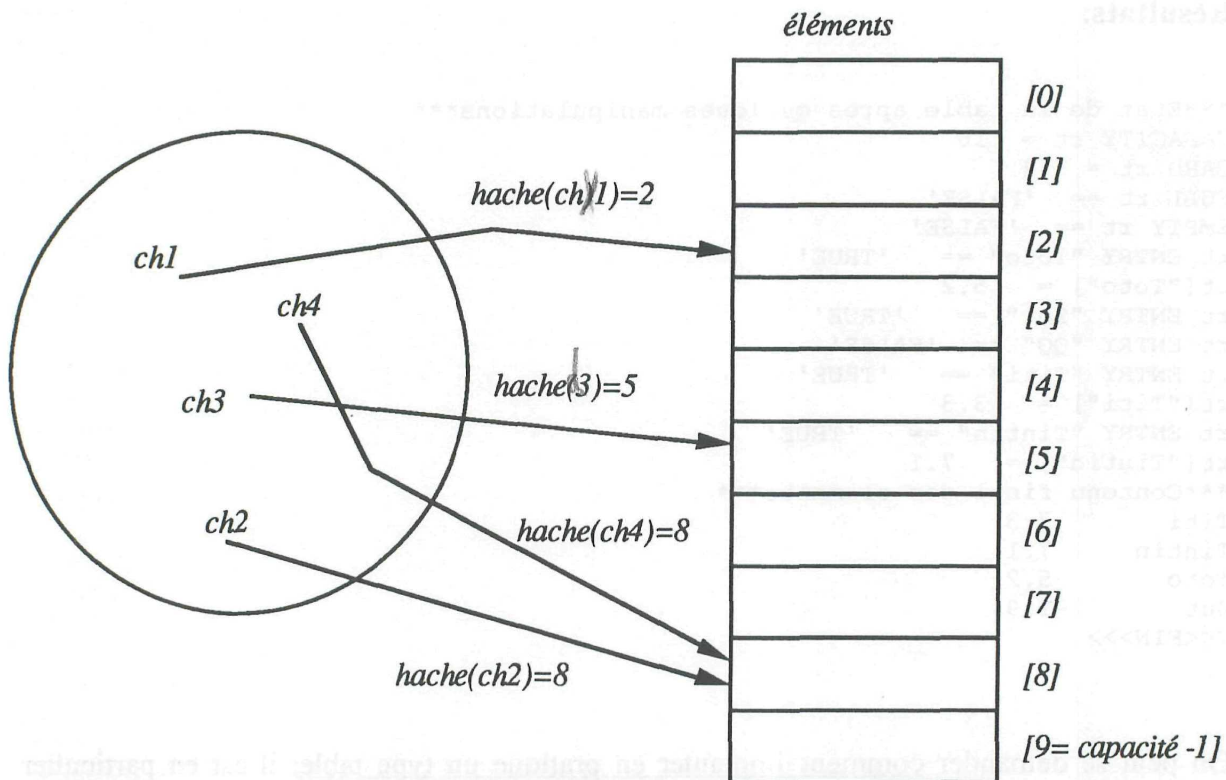


Figure 14

Dans la figure 14, on a supposé que l'on cherchait à représenter une table d'une capacité de dix éléments au moyen d'une fonction de hachage; de plus pour les quatre premières chaînes utilisées comme indices, cette fonction a livré les valeurs successives *hache (ch1) = 2*, *hache (ch2) = 8*, *hache (ch3) = 5* et *hache (ch4) = 8*. Il y a donc collision entre les composantes indicées par les chaînes *ch 2* et *ch4*: si l'on ne prend aucune mesure, ces deux composantes seront représentées au moyen de la même entrée *éléments [8]*. Il s'ensuit que toute implantation d'une table au moyen d'une fonction de hachage doit prévoir la possibilité de collision et doit gérer correctement les entrées en collision. Il existe plusieurs moyens de traiter les collisions; les différents moyens ont chacun leurs avantages et inconvénients. De toute manière, l'implantation en sera alourdie de façon non négligeable. On peut citer, en particulier, les techniques suivantes:

- Les entrées en collision sont insérées dans une rangée auxiliaire.
- En cas de collision, la rangée représentative est parcourue circulairement, élément par élément: la composante en collision est introduite dans la première entrée libre trouvée.
- En cas de collision, on procède à un hachage multiple: il est appliqué à la chaîne servant d'indice à une deuxième, puis si nécessaire une troisième, une quatrième ... fonction de hachage jusqu'à ce que l'on trouve une entrée libre dans la rangée représentative.
- Les entrées en collision sont chaînées en liste dont la tête est implantée dans l'entrée appropriée de la rangée représentative.

La choix de l'une ou l'autre de ces techniques dépendra, entre autres, du catalogue des opérations qui seront implantées sur la table concernée. En particulier, seule la représentation au moyen de listes permet de programmer, de manière relativement simple, l'élimination d'une composante de la table. Dans le cas des autres techniques que l'on a citées, il ne suffit pas de libérer l'entrée correspondante de la rangée représentative; il est de plus nécessaire de vérifier si une autre composante, stockée ailleurs à la suite d'une collision, ne doit pas réoccuper l'entrée ainsi libérée. Ceci implique un réhachage partiel ou complet de la table (ou, le cas échéant, des entrées stockées dans la rangée auxiliaire). Il s'ensuit que ce destructeur pourra être coûteux.

Quelle que soit la technique retenue, il faudra pouvoir discerner si une entrée est libre ou occupée et, dans ce dernier cas, si elle est occupée par la composante indicée par une chaîne donnée ou si elle est en collision avec cette dernière. Dans le cas du chaînage en une liste des composantes en collision, chaque noeud de cette liste comportera, en plus d'un élément de la

table et d'un lien au noeud suivant, la chaîne qui indice cet élément. Une entrée vide de la rangée sur laquelle débouche la fonction de hachage sera évidemment caractérisée par la valeur **nil**. Le principal inconvénient de cette technique est quelle est relativement coûteuse en mémoire (trois informations à stocker pour chaque composante de la table en plus de la rangée de têtes de listes sur laquelle débouche la fonction de hachage); pour le reste, elle présente beaucoup d'avantages.

Dans le cas des autres techniques que l'on a citées, il faut en tout cas prévoir une deuxième rangée *clés*, gérée par le même ensemble d'indices entiers que la rangée *éléments* dans laquelle seront stockées les chaînes indiçant les éléments correspondants de la table. Il faut, de plus, prévoir un indicateur qui permette de savoir si l'entrée est libre; si l'on connaît à priori une chaîne ω qui ne sera jamais utilisée comme indice d'une composante de la table, il est possible, de se passer de cet indicateur. Tous les éléments de la rangée *clés* seront initialisés à la valeur ω: cette valeur caractérisera une entrée libre. La figure 15 illustre la situation esquissé dans la figure 14; selon la deuxième technique proposée, on a placé la composante d'indice *ch 4* à la suite de celle d'indice *ch2* avec laquelle elle est entrée en collision.

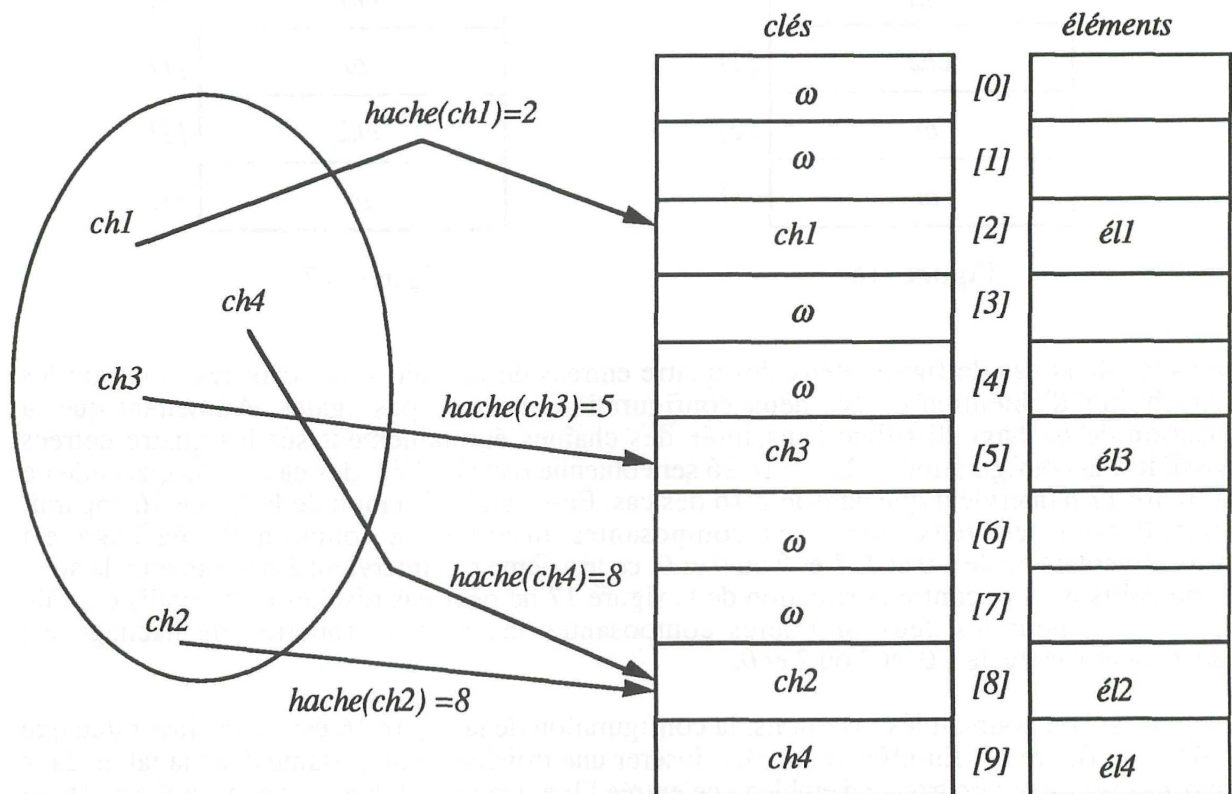


Figure 15

La technique faisant appel à une rangée auxiliaire (en réalité une paire de rangées auxiliaires) pour le stockage des composantes en collision n'est pas facile à mettre en oeuvre. En particulier, comment faut-il dimensionner la rangée auxiliaire relativement à la capacité de la rangée principale? Comment faut-il structurer cette rangée auxiliaire? A priori, cette technique ne paraît intéressante que si l'on ne s'attend qu'à une très faible proportion d'entrées en collision, ce qui n'est en général guère possible de prévoir.

La deuxième technique est assez simple à mettre en oeuvre. Si l'on doit implanter le destructeur, ce dernier n'exige qu'un rehachage partiel de la table: il reste donc relativement abordable. En-effet, seules sont susceptibles d'être rehachées les composantes stockées entre l'entrée que l'on a libéré et la prochaine composante vide. Ainsi, dans le cas représenté à la figure 15, si l'on veut supprimer l'entrée indicée par la chaîne *ch(2)*, on commence par considérer l'entrée suivante d'indice *ch4*; en recalculant la fonction de hachage pour cette dernière chaîne, on constate que la composante correspondante était en collision avec celle que l'on a supprimé. Il est donc

nécessaire de déplacer la composante d'indice *ch4* à la place de la composante que l'on a éliminé. Continuant le parcours circulaire de la rangée *clés* , on remarque que l'élément suivant *clés [0]* = ω : ceci implique qu'aucune autre composante n'était en collision avec celles d'indices *ch2* et *ch4* que l'on a supprimé ou déplacé. Il suffit donc de libérer l'emplacement occupé initialement par la composante d'indice *ch4* en stockant ω en *clés[9]*. Le principal inconvénient de cette technique est que, même avec une excellente fonction de hachage, elle ne répartit pas de manière uniforme les composantes dans les rangées représentatives; elle tend à favoriser l'apparition de longues séquences d'entrées non libres consécutives. En cas de collision, ceci dégrade de manière non négligeable l'efficacité du système. On peut voire ce manque d'uniformité déjà dans le cas d'une table de capacité égale à quatre. Pour cela, on considère les situations représentées aux figures 16 et 17 (dans lesquelles on ne fait figurer que la rangée *clés*).

clés	
ch1	*[0]*
ch2	*[1]*
ω	*[2]*
ω	*[3]*

Figure 16

clés	
ch1	*[0]*
ω	*[1]*
ch2	*[2]*
ω	*[3]*

Figure 17

Dans les deux cas de figure, deux des quatre entrées de la table sont occupées; pourtant les probabilités d'obtention de ces deux configurations ne sont pas égales. Admettant que la fonction de hachage distribue l'ensemble des chaînes équitablement sur les quatre entrées possibles, la configuration de la figure 16 sera obtenue dans le *3/16* des cas tandis que celle de la figure 17 n'intervient que dans le *2/16* des cas. En-effet, la situation de la figure 16 apparaît lorsque pour les deux premières composantes insérées, la fonction de hachage est successivement égale à *0* et *1*, *1* et *0* ou *0* et *0*; ce troisième cas intervient évidemment à la suite d'une collision. Par contre la situation de la figure 17 ne peut pas résulter d'une collision: elle apparaît si, pour les deux premières composantes insérées, la fonction de hachage est successivement égale à *0* et *2* ou *2* et *0*.

De plus, si l'on poursuit les insertions, la configuration de la figure 16 est moins favorable que celle de la figure 17. En-effet si l'on doit insérer une troisième composante dans la table, dans deux cas sur quatre on trouve d'emblée une entrée libre que ce soit dans le cas de la figure 16 ou celui de la figure 17; en cas de collision, par contre, il suffit toujours d'examiner une entrée supplémentaire dans le cas de la figure 17, mais on a une chance sur deux de devoir en examiner deux dans la situation représentée à la figure 16. On en déduit que dans ce dernier cas, il faudra en moyenne explorer *1.75* entrées avant de pouvoir insérer la troisième entrée; cette moyenne est par contre égale à *1.5* dans le cas plus favorable de la figure 17.

Soit r le taux de remplissage de la table, c'est-à-dire le rapport entre son nombre de composantes et sa capacité; on peut montrer que pour insérer une nouvelle composante dans la table, il faut en moyenne examiner un nombre d'éléments de l'ordre de *1/(1 - r) ** 2* lorsque l'on utilise cette technique d'implantation. Par contre pour retrouver une composante déjà présente dans la table, il suffit en moyenne d'examiner *1 / (1 - r)* éléments environ. Si les entrées libres et occupées étaient réparties de manière uniforme dans les rangées représentatives, on obtiendrait des comportements asymptotiques analogues, mais se dégradent moins vite lorsque le taux de remplissage s'approche de l'unité. Le nombre moyen d'éléments à visiter pour insérer une nouvelle composante serait alors de l'ordre de *1 / (1 - r)*; il serait de l'ordre de *-ln (1 - r)* pour retrouver une composante déjà dans la table.

C'est pour se rapprocher de ce dernier comportement que Knuth a proposé de faire un double hachage des entrées en collision. A ce sujet, il convient d'emblée de remarquer qu'une résolution de collisions au moyen d'un double hachage, voire d'un multiple hachage, doit être évitée s'il est nécessaire d'implanter le destructeur: celui-ci impliquerait en-effet un rehachage complet de la table et serait donc très coûteux. En cas de hachage multiple, on peut se demander comment choisir les fonctions de hachage successives: la chose n'est pas triviale si l'on veut que les résultats de ces fonctions soient indépendants les uns des autres en plus du fait qu'elles doivent être rapides et bien disperser les entrées.

On va maintenant illustrer une technique de double hachage; pour simplifier la création des deux fonctions de hachage, on va supposer que *capacité * (capacité - 1)* ne dépasse pas *integer* **max**. On commence alors par construire une seule fonction de hachage qui répartit les chaînes sur les entiers inclus entre *0* et *capacité * (capacité - 1) - 1*; soit ***h*** un tel entier: on calcule le quotient et le reste de la division de *h* par *capacité*. Le reste *h \ capacité* est une valeur entière comprise entre *0* et *capacité - 1* ; ce reste sera utilisé comme première fonction de hachage: il désigne la première entrée des rangées représentatives qui sera examinée. Le quotient augmenté de *1*, soit *1 + h % capacité* , est une valeur entière comprise entre *1* et *capacité - 1*; cette valeur sera utilisée comme deuxième fonction de hachage: elle désigne un incrément avec lequel les entrées des rangées représentatives seront explorées en cas de collision. Un problème intervient lorsque l'incrément et la capacité ne sont pas premiers entre eux: si l'on ne prend pas de précautions, on ne va explorer, en cas de collision, qu'un sous-ensemble des entrées des rangées représentatives.

Exemples:

capacité = 12
*capacité * (capacité -1) = 132*

On suppose que la fonction de hachage primitive a produit le résultat *h = 29*; comme première fonction de hachage, on utilise le reste *29 \ 12 = 5:* c'est l'entrée représentée en *clés [5]* et *éléments [5]* qui sera examinée en premier. Comme deuxième fonction de hachage, on utilise la valeur *1 + 29 % 12 = 3*: en cas de collision, les rangées représentatives seront explorées en incrémentant leurs indices par pas de *3* modulo *12*. On constate qu'on a alors les indices successifs:

5, 8, 11, 2

A ce stade, on retrouverait l'indice *5*; on n'aura donc exploré que quatre des douze entrées. Pour remédier à cet inconvénient, il faut se rappeler l'indice duquel on est parti; si l'on retombe sur cet indice sans avoir parcouru toutes les entrées, on augmente l'indice de 1 (et on se rappelle sa valeur). On peut montrer que cette procédure garantit que l'on explorera toutes les entrées de rangées représentatives; dans l'exemple qui nous occupe, les entrées seront examinées dans l'ordre suivant:

5, 8, 11, 2, (5)
6, 9, 0, 3, (6)
7, 10, 1, 4, (7)

Les indices soulignés sont ceux dont il faut se rappeler et incrémenter de *1* lorsqu'on retombe dessus. On vérifie, dans le cas présent, que l'on explorera bien, si nécessaire, toutes les entrées des rangées représentatives.

On va montrer, par l'exemple, cette manière d'implanter une table. Comme application, on veut automatiser la gestion d'un ensemble de comptes bancaires; chaque opération sera spécifiée au moyen d'une commande de la forme suivante:

> commande = opération nom prénom compte montant;
> opération = mot;
> prénom = mot;
> compte = nombre;
> montant = nombre;
> mot = lettre [{lettre |-|_}lettre];
> nombre = chiffre [{chiffre |_} chiffre];
> lettre = *A*| *B* | *C* | ...*Z* | *a* | *b* | *c* ...*z*;
> chiffre = *0* | *1* | *2* | ... *9*.

Quatre commandes seront acceptées par le système; pour chaque commande, il est indiqué le nom et le prénom du possesseur du compte, le numéro du compte ainsi qu'une somme susceptible d'être déposée dans le compte ou en être retirée. Les noms de commande peuvent être écrits arbitrairement en majuscule ou en minuscules.

OUVRIR	ouverture d'un nouveau compte et dépôt initial du montant donné.
DEPOSER	dépôt du montant donné dans un compte existant.
CONSULTER	consulter si le montant donné peut être retiré du compte donné.
RETIRER	retirer le montant donné du compte donné.

Ci-dessous figure une suite de commandes.

```
ouvrir Pinar Srahlec 086_139_713 25
ouvrir Pinar Srahlec 086_139_713 80
consulter Pinar Srahlec 086_139_713
consulter Pinar Srahlec 086_139_712 55
consulter Pianr Srahlec 086_139_713 55
retirer Pinar Srahlec 086_139_713 55
fermer Pinar Srahlec 086_139_713 80
ouvrir Zoom la_Terreur 999_888_777  1_000_000_000
Ouvrir  le_Grand Louis_XIV  686_702_904  1_999_999_999_999
Ouvrir le_Grand Louis_XIV   686702904 1_999_999_999
ouvrir Satan Adolphe 576_832_926 1_234_000
ouvrir Martin Pecheur 402_712_808  1024
ouvrir Zombie le_Fou  839_112_331  999
retirer le_Grand Louis_XIV  686_702_904 987_654_321
ouvrir le_Roy Charles 702_646_135 880_901
ouvrir  du_Terroir  Julie 002_602_620 137
retirer Zoom la Terreur 999_888_777 145_802_996
retirer Zoom la_Terreur 999_888_777 145_802_996
ouvrir Pinar Srahlec 086139713 1_000_000
ouvrir Pinar Srahlec 086139714 1_000_000
 consulter Zombie le_Fou  839112331 1000
deposer Satan Adolphe 576832926 490127801
deposer Satan Adolphe 576_832_926 490127801
 ouvrir la_Blanche Louise  001_834_172  5800
retirer la_Blanche  Louise  001834172  4310
OUVRIR  dans-la-Lune Pierrot  996702128  0
ouvrir dans-la-Lune Pierrot 996702128 100
consulter le_Roy Charles 702_646_135 1_086
retirer le_roy Charles 702_646_135 1_086
ouvrir Zebulon Claudy 635_098_777 10
deposer du_Terroir Julie  002_602_620  41
ouvrir Knight Chevalin 661_772_883 739_981
deposer dans-la-Lune Pierrot  996702128 137
ouvrir la_Bete a_Bon_Dieu 304_946_825  1000
deposer  le_Roy  Charles  702_646_135  279
ouvrir de_Bidon-Ville  Julot  444_126_349  33
ouvrir de_Bidon-Ville  Julot  444_126_349  50
  RETIRER la_Bete a_Bon_Dieu  304_946_825 186
deposer Srahlec Pinar 086_139_713 126
deposer Pinar  Srahlec 086_139_713   126
  OUVRIR  de-Medicis  Cathy  001_112_223  102_086
deposer Zoom la_Terreur  999_888_777  201_726_844
ouvrir du_Paradis Oiseau  331_709_861 5000
  deposer du_Paradis Oiseau  331_709_861 1
ouvrir  la_Belle Cleopatre  682_711_096   754_833_111
consulter la_belle Cleopatre 682_711_096  602_777_123
 consulter la_Belle Cleopatre  682_711_096  602_777_123
retirer la_Belle Cleopatre 682_711_096  602777123
  Retirer Pinar Srahlec 086_139_713 87_133
  Retirer Pinar Srahlec 086_139_714 87_133
deposer de-Medicis Cathy 001_112_223  78_097
ouvrir du_Tonneau Silene 331113331 80808080
retirer du_Tonneau Silene 331113331 19191919
```

L'interprétation de ces commandes donne lieu aux résultats suivants.

```
*****Mutations sur les comptes bancaires*****

   1--->ouvrir Pinar Srahlec 086_139_713 25
***OUVRIR compte [Pinar, Srahlec: 086139713] montant              25Fr.***
   ###depot initial 25Fr. insuffisant###
   1--->mutation non effectuee

   2--->ouvrir Pinar Srahlec 086_139_713 80
***OUVRIR compte [Pinar, Srahlec: 086139713] montant              80Fr.***
   compte ouvert, credit initial:    80Fr.
   2--->mutation effectuee

   3--->consulter Pinar Srahlec 086_139_713
   3--->commande incorrecte

   4--->consulter Pinar Srahlec 086_139_712 55
***CONSULTER compte [Pinar, Srahlec: 086139712] montant              55Fr.***
   ###compte inexistant###
   4--->mutation non effectuee

   5--->consulter Pianr Srahlec 086_139_713 55
***CONSULTER compte [Pianr, Srahlec: 086139713] montant              55Fr.***
   ###compte inexistant###
   5--->mutation non effectuee

   6--->retirer Pinar Srahlec 086_139_713 55
***RETIRER compte [Pinar, Srahlec: 086139713] montant              55Fr.***
   ###retrait de            55Fr. impossible###
   credit            80Fr. insuffisant
   6--->mutation non effectuee

   7--->fermer Pinar Srahlec 086_139_713 80
***FERMER compte [Pinar, Srahlec: 086139713] montant              80Fr.***
   ###operation inconnue###
   7--->mutation non effectuee

   8--->ouvrir Zoom la_Terreur 999_888_777  1_000_000_000
***OUVRIR compte [Zoom, la_Terreur: 999888777] montant  1000000000Fr.***
   compte ouvert, credit initial:   1000000000Fr.
   8--->mutation effectuee

   9--->Ouvrir  le_Grand Louis_XIV  686_702_904  1_999_999_999_999
   9--->commande incorrecte

  10--->Ouvrir le_Grand Louis_XIV   686702904 1_999_999_999
***OUVRIR compte [le_Grand, Louis_XIV: 686702904] montant
1999999999Fr.***
  compte ouvert, credit initial:  1999999999Fr.
  10--->mutation effectuee

  11--->ouvrir Satan Adolphe 576_832_926 1_234_000
***OUVRIR compte [Satan, Adolphe: 576832926] montant        1234000Fr.***
  compte ouvert, credit initial:    1234000Fr.
  11--->mutation effectuee

  12--->ouvrir Martin Pecheur 402_712_808  1024
***OUVRIR compte [Martin, Pecheur: 402712808] montant           1024Fr.***
  compte ouvert, credit initial:          1024Fr.
  12--->mutation effectuee
```

```
  13--->ouvrir Zombie le_Fou  839_112_331  999
***OUVRIR compte [Zombie, le_Fou: 839112331] montant          999Fr.***
   compte ouvert, credit initial:          999Fr.
  13--->mutation effectuee

  14--->retirer le_Grand Louis_XIV  686_702_904 987_654_321
***RETIRER compte [le_Grand, Louis_XIV: 686702904] montant
987654321Fr.***
   retrait de    987654321Fr. effectue
   solde:   1012345678Fr.
  14--->mutation effectuee

  15--->ouvrir le_Roy Charles 702_646_135 880_901
***OUVRIR compte [le_Roy, Charles: 702646135] montant        880901Fr.***
   compte ouvert, credit initial:        880901Fr.
  15--->mutation effectuee

  16--->ouvrir  du_Terroir  Julie 002_602_620 137
***OUVRIR compte [du_Terroir, Julie: 002602620] montant           137Fr.***
   compte ouvert, credit initial:          137Fr.
  16--->mutation effectuee

  17--->retirer Zoom la Terreur 999_888_777 145_802_996
  17--->commande incorrecte

  18--->retirer Zoom la_Terreur 999_888_777 145_802_996
***RETIRER compte [Zoom, la_Terreur: 999888777] montant    145802996Fr.***
   retrait de    145802996Fr. effectue
   solde:    854197004Fr.
  18--->mutation effectuee

  19--->ouvrir Pinar Srahlec 086139713 1_000_000
***OUVRIR compte [Pinar, Srahlec: 086139713] montant      1000000Fr.***
   ###compte existe deja###
  19--->mutation non effectuee

  20--->ouvrir Pinar Srahlec 086139714 1_000_000
***OUVRIR compte [Pinar, Srahlec: 086139714] montant      1000000Fr.***
   compte ouvert, credit initial:  1000000Fr.
  20--->mutation effectuee

  21---> consulter Zombie le_Fou  839112331 1000
***CONSULTER compte [Zombie, le_Fou: 839112331] montant          1000Fr.***
   credit actuel:          999Fr.
   retrait de         1000Fr. impossible; montant disponible
949Fr.
  21--->mutation effectuee

  22--->deposer Satan Adolphe 576832926 490127801
***DEPOSER compte [Satan, Adolphe: 576832926] montant    490127801Fr.***
      490127801Fr. deposes; nouveau credit:   491361801Fr.
  22--->mutation effectuee

  23--->deposer Satan Adolphe 576_832_926 490127801
***DEPOSER compte [Satan, Adolphe: 576832926] montant    490127801Fr.***
      490127801Fr. deposes; nouveau credit:    981489602Fr.
  23--->mutation effectuee

  24---> ouvrir la_Blanche Louise  001_834_172  5800
***OUVRIR compte [la_Blanche, Louise: 001834172] montant          5800Fr.***
   compte ouvert, credit initial:        5800Fr.
  24--->mutation effectuee
```

```
  25--->retirer la_Blanche  Louise 001834172  4310
***RETIRER compte [la_Blanche, Louise: 001834172] montant
4310Fr.***
   retrait de         4310Fr. effectue
   solde:          1490Fr.
  25--->mutation effectuee

  26--->OUVRIR  dans-la-Lune Pierrot  996702128  0
  26--->commande incorrecte

  27--->ouvrir dans-la-Lune Pierrot 996702128 100
***OUVRIR compte [dans-la-Lune, Pierrot: 996702128] montant
100Fr.***
   compte ouvert, credit initial:          100Fr.
  27--->mutation effectuee

  28--->consulter le_Roy Charles 702_646_135 1_086
***CONSULTER compte [le_Roy, Charles: 702646135] montant        1086Fr.***
   credit actuel:        880901Fr.
   retrait de         1086Fr. possible
  28--->mutation effectuee

  29--->retirer le_roy Charles 702_646_135 1_086
***RETIRER compte [le_roy, Charles: 702646135] montant         1086Fr.***
   ###compte inexistant###
  29--->mutation non effectuee

  30--->ouvrir Zebulon Claudy 635_098_777 10
***OUVRIR compte [Zebulon, Claudy: 635098777] montant          10Fr.***
   ###depot initial 10Fr. insuffisant###
  30--->mutation non effectuee

  31--->deposer du_Terroir Julie  002_602_620  41
***DEPOSER compte [du_Terroir, Julie: 002602620] montant          41Fr.***
          41Fr. deposes; nouveau credit:          178Fr.
  31--->mutation effectuee

  32--->ouvrir Knight Chevalin 661_772_883 739_981
***OUVRIR compte [Knight, Chevalin: 661772883] montant        739981Fr.***
   compte ouvert, credit initial:       739981Fr.
  32--->mutation effectuee

  33--->deposer dans-la-Lune Pierrot  996702128 137
***DEPOSER compte [dans-la-Lune, Pierrot: 996702128] montant
137Fr.***
         137Fr. deposes; nouveau credit:          237Fr.
  33--->mutation effectuee

  34--->ouvrir la_Bete a_Bon_Dieu 304_946_825  1000
***OUVRIR compte [la_Bete, a_Bon_Dieu: 304946825] montant
1000Fr.***
   compte ouvert, credit initial:         1000Fr.
  34--->mutation effectuee

  35--->deposer  le_Roy  Charles  702_646_135  279
***DEPOSER compte [le_Roy, Charles: 702646135] montant          279Fr.***
         279Fr. deposes; nouveau credit:       881180Fr.
  35--->mutation effectuee

  36--->ouvrir de_Bidon-Ville  Julot  444_126_349  33
***OUVRIR compte [de_Bidon-Ville, Julot: 444126349] montant
33Fr.***
   ###depot initial 33Fr. insuffisant###
```

```
  36--->mutation non effectuee

  37--->ouvrir de_Bidon-Ville  Julot  444_126_349  50
***OUVRIR compte [de_Bidon-Ville, Julot: 444126349] montant
50Fr.***
   compte ouvert, credit initial:           50Fr.
  37--->mutation effectuee

  38--->  RETIRER la_Bete a_Bon_Dieu  304_946_825 186
***RETIRER compte [la_Bete, a_Bon_Dieu: 304946825] montant
186Fr.***
   retrait de         186Fr. effectue
   solde:          814Fr.
  38--->mutation effectuee

  39--->deposer Srahlec Pinar 086_139_713 126
***DEPOSER compte [Srahlec, Pinar: 086139713] montant          126Fr.***
   ###compte inexistant###
  39--->mutation non effectuee

  40--->deposer Pinar  Srahlec 086_139_713   126
***DEPOSER compte [Pinar, Srahlec: 086139713] montant          126Fr.***
         126Fr. deposes; nouveau credit:          206Fr.
  40--->mutation effectuee

  41--->  OUVRIR  de-Medicis  Cathy  001_112_223  102_086
***OUVRIR compte [de-Medicis, Cathy: 001112223] montant       102086Fr.***
   compte ouvert, credit initial:       102086Fr.
  41--->mutation effectuee

  42--->deposer Zoom la_Terreur  999_888_777  201_726_844
***DEPOSER compte [Zoom, la_Terreur: 999888777] montant    201726844Fr.***
      201726844Fr. deposes; nouveau credit:   1055923848Fr.
  42--->mutation effectuee

  43--->ouvrir du_Paradis Oiseau  331_709_861 5000
***OUVRIR compte [du_Paradis, Oiseau: 331709861] montant          5000Fr.***
   compte ouvert, credit initial:         5000Fr.
  43--->mutation effectuee

  44--->  deposer du_Paradis Oiseau  331_709_861 1
***DEPOSER compte [du_Paradis, Oiseau: 331709861] montant
1Fr.***
             1Fr. deposes; nouveau credit:         5001Fr.
  44--->mutation effectuee

  45--->ouvrir  la_Belle Cleopatre  682_711_096   754_833_111
***OUVRIR compte [la_Belle, Cleopatre: 682711096] montant
754833111Fr.***
   compte ouvert, credit initial:    754833111Fr.
  45--->mutation effectuee

  46--->consulter la_belle Cleopatre 682_711_096  602_777_123
***CONSULTER compte [la_belle, Cleopatre: 682711096] montant
602777123Fr.***
   ###compte inexistant###
  46--->mutation non effectuee

  47---> consulter la_Belle Cleopatre  682_711_096  602_777_123
***CONSULTER compte [la_Belle, Cleopatre: 682711096] montant
602777123Fr.***
   credit actuel:    754833111Fr.
   retrait de    602777123Fr. possible
  47--->mutation effectuee
```

```
  48--->retirer la_Belle Cleopatre 682_711_096  602777123
***RETIRER compte [la_Belle, Cleopatre: 682711096] montant
602777123Fr.***
   retrait de    602777123Fr. effectue
   solde:    152055988Fr.
  48--->mutation effectuee

  49--->  Retirer Pinar Srahlec 086_139_713 87_133
***RETIRER compte [Pinar, Srahlec: 086139713] montant         87133Fr.***
   ###retrait de         87133Fr. impossible###
   credit           206Fr. insuffisant
  49--->mutation non effectuee

  50--->  Retirer Pinar Srahlec 086_139_714 87_133
***RETIRER compte [Pinar, Srahlec: 086139714] montant         87133Fr.***
   retrait de         87133Fr. effectue
   solde:       912867Fr.
  50--->mutation effectuee

  51--->deposer de-Medicis Cathy 001_112_223 78_097
***DEPOSER compte [de-Medicis, Cathy: 001112223] montant         78097Fr.***
        78097Fr. deposes; nouveau credit:     180183Fr.
  51--->mutation effectuee

  52--->ouvrir du_Tonneau Silene 331113331 80808080
***OUVRIR compte [du_Tonneau, Silene: 331113331] montant      80808080Fr.***
   compte ouvert, credit initial:     80808080Fr.
  52--->mutation effectuee

  53--->retirer du_Tonneau Silene 331113331 19191919
***RETIRER compte [du_Tonneau, Silene: 331113331] montant
19191919Fr.***
   retrait de     19191919Fr. effectue
   solde:     61616161Fr.
  53--->mutation effectuee

*****Fin des mutations*****


*****Etat final des comptes*****
[de_Bidon-Ville, Julot: 444126349]           50Fr.
[du_Paradis, Oiseau: 331709861]              5001Fr.
[de-Medicis, Cathy: 001112223]             180183Fr.
[Martin, Pecheur: 402712808]               1024Fr.
[le_Grand, Louis_XIV: 686702904]       1012345678Fr.
[Knight, Chevalin: 661772883]              739981Fr.
[Satan, Adolphe: 576832926]             981489602Fr.
[du_Tonneau, Silene: 331113331]          61616161Fr.
[Zoom, la_Terreur: 999888777]         1055923848Fr.
[Zombie, le_Fou: 839112331]                 999Fr.
[la_Belle, Cleopatre: 682711096]        152055988Fr.
[dans-la-Lune, Pierrot: 996702128]          237Fr.
[la_Blanche, Louise: 001834172]            1490Fr.
[le_Roy, Charles: 702646135]             881180Fr.
[Pinar, Srahlec: 086139714]              912867Fr.
[du_Terroir, Julie: 002602620]              178Fr.
[la_Bete, a_Bon_Dieu: 304946825]            814Fr.
[Zebulon, Claudy: 635098777]         ###compte vide###
[Pinar, Srahlec: 086139713]                 206Fr.
<<<FIN>>>
```

On constate que chaque mutation donne lieu à un protocole; ce dernier indique si l'opération correspondante a été effectuée ou si elle s'est révélée impossible. Une opération peut être refusée soit parce qu'elle ne satisfait pas à la syntaxe requise, soit parce que l'opération correspondante n'a pas été définie, soit pour d'autres raisons de nature sémantique: ouverture d'un compte existant déjà, dépôt, consultation ou retrait d'un compte inexistant, retrait d'une somme trop forte (à ce dernier propos, on constate qu'un compte, une fois ouvert, doit toujours contenir au minimum *50 Fr.*)
A la fin de son exécution, le système imprime (dans un ordre arbitraire) l'état de chacun des comptes.

Le programme *credit_suisse* est relativement conséquent; il est cependant décomposé en un certain nombre de parties relativement simples. Sa structure est indiquée dans la figure 18; bien entendu, dans ce schéma, seules les déclarations les plus imposantes ont été citées.

La partie exécutable du programme *credit_suisse* est un interprète de commandes; pour cela, il s'appuye notamment sur les opérations *cherche_mot, cherche_nombre* et *cherche_positif.*

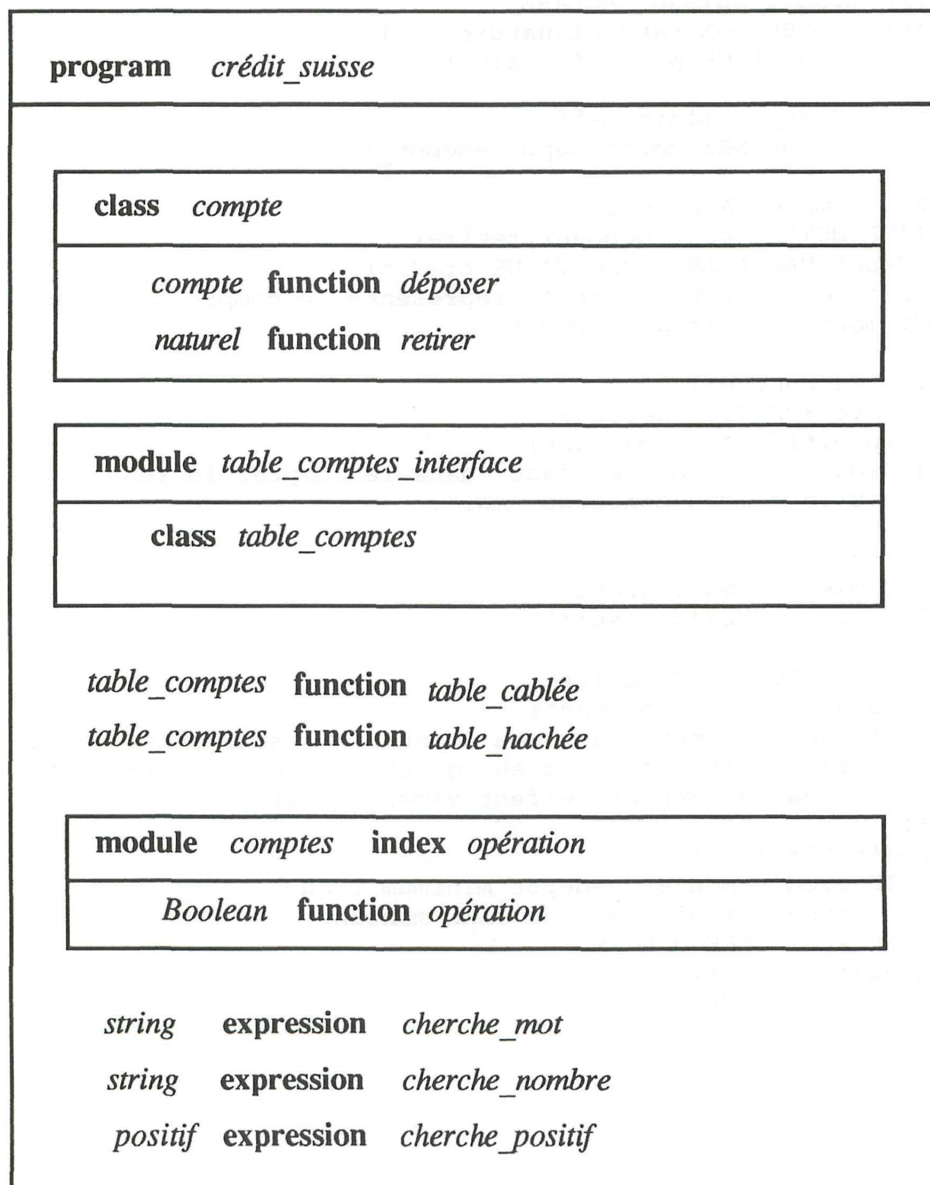


Figure 18

Une fois une commande analysée et acceptée, elle est communiquée, au moyen de sa fonction d'indiçage *opération*, au module *comptes*; c'est ce dernier qui exécute la commande. Pour cela, il dispose d'une table extensible *clients* du type *table_comptes*. Les tables du type *table_comptes* ont été implantées selon la technique indiquée au chapitre 5; les objets correspondants sont créés par l'intermédiaire de fonctions génératrices. Il a été fourni deux fonctions génératrices: la première *table_câblée* implante la table au moyen d'une table prédéfinie du langage Newton tandis que la seconde *table_hachée* fait appel à une paire de rangées *clés* et *éléments* gérées au moyen d'un double hachage. Jusqu'à (et y compris) la classe *compte* , ce programme ne nécessite pas de commentaire particulier.

```
credit_suisse                                        Vax Newton Compiler 0.2c1
Page    1
                       Source listing

/* /*OLDSOURCE=USER2:[RAPIN]CREDIT.NEW*/ */
PROGRAM credit_suisse DECLARE
  integer SUBRANGE naturel(naturel>=0)
          SUBRANGE positif(positif>0);

  CONSTANT depot_minimum=50;
  positif SUBRANGE depot(depot>=depot_minimum);

  CLASS compte VALUE moi
    ATTRIBUTE credit,deposer,retirer
    (depot VARIABLE somme VALUE credit)
  (*Un objet du type  compte  represente un compte bancaire
    de montant initial  credit .
  *)
  DECLARE(*compte*)
    compte FUNCTION deposer
      (positif VALUE montant)
    (*Depose la somme  montant  dans le compte; le resultat
      est le compte concerne  moi .
    *)
    DO(*deposer*)
      somme:=somme+montant
    TAKE moi DONE(*deposer*);

    naturel FUNCTION retirer
      (positif VALUE montant)
    (*Retire du compte la somme  montant  ; sans effet si le
      solde serait inferieur au  depot_minimum. Le resultat
      est egal au montant effectivement retire.
    *)
    DO(*retirer*)TAKE
      IF credit-montant>=depot_minimum THEN
        somme:=credit-montant TAKE montant
      DEFAULT TAKE 0 DONE
    DONE(*retirer*)
  DO(*compte*)DONE;
  /* /*EJECT*/ */
```

```
MODULE table_comptes_interface
  ATTRIBUTE interrogateur,denombreur,chercheur,selecteur,
            action,iterateur,table_comptes
DECLARE(*table_comptes_interface*)
  Boolean FUNCTOR interrogateur;
  naturel FUNCTOR denombreur;
  Boolean FUNCTOR chercheur(string VALUE cle);
  compte ACCESSOR selecteur(string VALUE cle);
  ACTOR action(string VALUE cle; compte REFERENCE var);
  ACTOR iterateur(action VALUE acte);

  CLASS table_comptes VALUE moi
    INDEX element
    ATTRIBUTE vide,plein,capacite,quantite,entree,parcourir
    (positif VALUE capacite;
     interrogateur VALUE vd,pln;
     denombreur VALUE quant;
     chercheur VALUE entr;
     selecteur VALUE elt;
     iterateur VALUE parc)
   (*Cette classe est l'interface necessaire a l'etablissement
    d'une table associative d'au maximum  capacite  variables
    du type  compte . Les tables individuelles seront crees
    par l'intermediaire de fonctions generatrices; l'implanteur
    fournira les objets proceduraux suivants:
      vd      pour implanter vide
      pln          "         plein
      quant        "         quantite
      entr         "         entree
      elt          "         element
      parc         "         parcourir
  *)
  DECLARE(*table_comptes*)
    Boolean EXPRESSION
      vide=vd[](*vrai ssi la table est vide*),
      plein=pln[](*vrai ssi la table est pleine*);

    naturel EXPRESSION
      quantite=quant[](*le nombre d'elements de la table*);

    Boolean FUNCTION entree
      (string VALUE cle)
     (*vrai ssi la table possede une entree indicee par la
      chaine  cle .
    *)
    DO TAKE entr[cle] DONE(*entree*);

    compte ACCESS element
      (string VALUE cle)
     (*la variable indicee par la chaine  cle ; insere dans
      la table une telle composante si elle n'en possede pas
      deja une.
```

```
credit_suisse                                   Vax Newton Compiler 0.2c1
Page    3
                    Source listing

        Condition d'emploi:  plein|>entree(cle)
      *)
      DO(*element*)
        UNLESS plein|>entree(cle) THEN
          print(line,"###Insertion dans table pleine###")
        RETURN DONE
      TAKE elt[cle] DONE(*element*);

      table_comptes FUNCTION parcourir
        (action VALUE acte)
      (*Parcourt, dans un ordre non specifie, les elements de
        la table. Pour chaque element, effectue l'enonce
        acte[cle,var]  dans lequel  cle  est la chaine qui
        indice l'element et  var  est la variable qui lui est
        associee. Le resultat est la table concernee  moi .
      *)
      DO parc[acte] TAKE moi DONE
    DO(*table_comptes*)DONE
  DO(*table_comptes_interface*)DONE;
    /* /*EJECT*/ */
```

Les objets du type *table_comptes* seront crées par l'intermédiaire de fonctions génératrices qui en fourniront une implémentation concrète. En plus de la classe protocole *table_comptes*, il est exporté du module *table_comptes_interface* les types d' objets procéduraux que les fonctions génératrices doivent livrer comme paramètres aux objets qu'elles engendrent.

Dans une version future du langage, on peut imaginer qu'un tel module pourra être compilé séparément une fois pour toutes. Ceci permettra aux implanteurs du type *table_comptes* et à ses utilisateurs de travailler indépendamment les uns les autres.

```
credit_suisse                                      Vax Newton Compiler 0.2c1
Page    4
                   Source listing

  299    table_comptes FUNCTION table_cablee
  302      (positif VALUE capacite)
  307    (*Fournit un objet de la classe  table_comptes  d'au maximum
  307      capacite  elements ; cet objet est implante au moyen d'une
  307      table predefinie.
  307    *)
  307    DECLARE(*table_cablee*)
  308      (*Representation interne*)
  308      compte TABLE comptable VALUE t=comptable(capacite);
  319
  319      (*Implantation des operations*)
  319      Boolean EXPRESSION vide=EMPTY t, plein=FULL t;
  331
  331      naturel EXPRESSION quantite=CARD t;
  338
  338      Boolean FUNCTION entree
  341        (string VALUE cle)
  346      DO TAKE t ENTRY cle DONE;
  353
  353      compte ACCESS composante
  356        (string VALUE cle)
  361      DO TAKE t[cle] DONE;
  369
  369      PROCEDURE parcours(action VALUE acte)DO
  377        THROUGH t INDEX cle REFERENCE element REPEAT
  384          acte[cle,element]
  390        REPETITION
  391      DONE(*parcours*)
  392    DO(*table_cablee*)TAKE
  394      (*implante la table*)
  394      table_comptes(capacite,
  398                    interrogateur(vide),interrogateur(plein),
  408                    denombreur(quantite),
  413                    chercheur(entree),selecteur(composante),
  423                    iterateur(parcours))
  428    DONE(*table_cablee*);
  430    /* /*EJECT*/ */
```

Cette première fonction génératrice *table_câblée* ne nécessite guère de commentaires; elle fournit une implantation concrète du type *table_câblée* au moyen d'une table *t* du type *compte* **table** *comptable*.

```
 430   table_comptes FUNCTION table_hachee
 433     (positif VALUE capacite; string VALUE omega)
 442   (*Cree un objet de la classe  table_comptes  d'au plus
 442     capacite elements; cet objet est implante au moyen d'une
 442     fonction de hachage. Les entrees de cet objet ne pourront
 442     pas etre indicees au moyen de la chaine  omega .
 442
 442     Condition d'emploi:
 442       capacite*(capacite-1)*(SUCC ORD character MAX)<=integer MAX
 442   *)
 442   DECLARE(*table_hachee*)
 443     (*representation interne*)
 443     string ROW strow VALUE cles=
 449       THROUGH strow(0 TO PRED capacite):=omega REPETITION;
 461     compte ROW crow VALUE elements=crow(0 TO PRED capacite);
 475
 475     naturel VARIABLE nombre:=0;
 481     (*le nombre d'elements de la table*)
 481
 481     string VARIABLE derniere_cle:=omega;
 487     (*la cle de la derniere entree cherchee dans la table*)
 487     compte REFERENCE dernier_element;
 491     (*repere a la derniere entree cherchee dans la table*)
 491     integer VARIABLE derniere_pos;
 495     (*apres recherche infructueuse d'un element, l'indice ou
 495       un tel element devrait, le cas echeant, etre insere
 495     *)
 495     Boolean VARIABLE present:=FALSE;
 501     (*vrai ssi la derniere recherche a produit un element de
 501       la table reflete par  derniere_cle  et  dernier_element
 501     *)
 501
 501     (*definitions et operations auxiliaires*)
 501     integer VALUE limite=capacite*(PRED capacite) MAX 1;
 514     integer VALUE base=SUCC ORD character MAX;
 523
 523     integer FUNCTION hache
 526       (string VALUE mot)
 531     (*cette fonction de hachage produit une valeur entiere
 531       comprise entre  0  et  limite-1
 531     *)
 531     DECLARE(*hache*)
 532       integer VARIABLE s:=(capacite+LENGTH mot)\limite
 544     DO(*hache*)
 545       THROUGH mot VALUE c REPEAT
 550         s:=(s*base+ORD c)\limite
 562       REPETITION
 563     TAKE s DONE(*hache*);
 567
 567     PROCEDURE cherche_mot
 569       (string VALUE mot)
 574     (*recherche si la table possede une composante d'indice
```

```
      mot:

      si oui, stocke TRUE en  present ; ajuste  derniere_cle
              et  dernier_element  pour refleter cet element.
      si non, stocke FALSE en  present ; stocke en  derniere_pos
              l'indice, dans les rangees representatives, ou
              l'element pourrait, le cas echeant, etre insere
              (cette derniere operation n'a pas lieu si la table
              est pleine)
    *)
    DECLARE(*cherche_mot*)
      integer VALUE
        hachage=hache(mot),
        pas_tabulation=SUCC hachage%capacite;
      (*0<=hachage/\hachage<limite
        1<=pas_tabulation/\pas_tabulation<capacite
      *)
      integer VARIABLE
        pos_init,pos:=hachage\capacite,
        compte:=0;
      (*0<=pos_init/\pos_init<capacite
        0<=pos/\pos<capacite
      *)
    DO(*cherche_mot*)
      CYCLE recherche REPEAT

      IF cles[pos]=omega THEN
        (*on tombe sur une place vide*)
        present:=FALSE; derniere_pos:=pos
      EXIT recherche DONE;

      IF cles[pos]=mot THEN
        (*on a trouve l'element requis*)
        present:=TRUE; derniere_cle:=mot;
        dernier_element->elements[pos]
      EXIT recherche DONE;

      IF (compte:=SUCC compte)=capacite THEN
        (*on a tout visite sans succes*)
        present:=FALSE
      EXIT recherche DONE;

        (*visite une autre entree*)
        IF
          (pos:=(pos+pas_tabulation)\capacite)=pos_init
        THEN
          pos:=(pos_init:=(SUCC pos_init)\capacite)
        DONE
      REPETITION(*CYCLE recherche*)
    DONE(*cherche_mot*)
  DO(*table_hachee*)TAKE
    (*livre l'implantation*)
```

```
      table_comptes
        (capacite,

         BODY interrogateur DO TAKE nombre=0 DONE,
         BODY interrogateur DO TAKE nombre=capacite DONE,

         BODY denombreur DO TAKE nombre DONE,

         BODY
           chercheur(string VALUE mot)
         DO TAKE
           IF mot=omega THEN
             FALSE ELSE
           IF present/\mot=derniere_cle THEN
             TRUE
           DEFAULT cherche_mot(mot) TAKE present DONE
         DONE(*chercheur*),

         BODY
           selecteur(string VALUE mot)
         DO
           IF mot=omega THEN
             print(line,"###table hachee indicee par chaine reservee"
                   _#"#,omega,#"# "###")
           RETURN DONE;
           (*l'interface a fait evaluer  entree , donc  cherche_mot ,
             lors du test de debordement; on peut donc utiliser
             present
           *)
           UNLESS present THEN
             cles[derniere_pos]:=(derniere_cle:=mot);
             dernier_element->elements[derniere_pos];
             nombre:=SUCC nombre; present:=TRUE
           DONE
         TAKE dernier_element DONE(*selecteur*),

         BODY
           iterateur(action VALUE acte)
         DO
           THROUGH
             cles INDEX pos VALUE mot
           REPEAT
             IF mot~=omega THEN
               acte[mot,elements[pos]]
             DONE
           REPETITION
         DONE(*iterateur*))
    DONE(*table_hachee*);
    /* /*EJECT*/ */
```

La fonction génératrice *table_hachée* fournit un bon modèle d'implantation des tables prédéfinies; ces dernières sont effectivement traitées avec un double hachage, méthode valable si le destructeur n'a pas à être implanté. Bien entendu, les tables prédéfinies sont gérées au moyen de routines efficaces en langage d'assemblage; notamment, les deux fonctions de hachage tiennent compte de la représentation interne des chaînes concernées. On constate qu'il est nécessaire de fournir, à la fonction *table_hachée* , une chaîne *omega* dont on sait à priori qu'elle ne sera pas utilisée comme indice d'une composante de la table. Il n'est en général pas difficile de trouver une telle chaîne; si nécessaire, il est toujours possible d'y inclure des caractères de contrôle.

La partie centrale de l'algorithme, utilisé pour chercher une entrée dans la table ou en insérer une, est inclus dans la procédure *cherche_mot* .

On peut supposer qu'assez souvent, on fera plusieurs utilisations successives de la même entrée de la table. Il peut être intéressant de court-circuiter, dans ce cas, le recours répété à la fonction de hachage et à l'algorithme de recherche qui s'ensuit. On a utilisé, à cette fin, les variables *présent* , *dernière_clé* et *dernière_pos* ainsi que le repère *dernier_élément*. Si *présent* est vraie, on sait que *dernier_élément* dénote une composante de la table indicée par la chaîne *dernière_clé*. Par contre, en cas de recherche infructueuse d'une composante, *présent* devient fausse et il est stocké en *dernière_pos* le numéro de l'entrée, dans les rangées représentatives *clés* et *éléments*, où cette composante pourrait être insérée. On peut constater la manière dont il est fait usage de ces informations dans le sélecteur.

```
credit_suisse                                    Vax Newton Compiler 0.2c1
Page    8
                  Source listing

MODULE comptes
  INDEX operation
DECLARE(*comptes*)
  (*Les comptes des clients de la banque concernee*)
  CONSTANT taille_initiale=15,rempli_min=.5,rempli_max=.8;
  table_comptes VARIABLE
  clients:=table_hachee(taille_initiale,"???");

  PROCEDURE refait_table_clients
    (positif VALUE nouvelle_capacite)
  DO(*refait_table_clients*)
    (table_cablee(nouvelle_capacite)=:clients).
      parcourir
        (BODY
           action(string VALUE client; compte REFERENCE cpt)
         DO clients[client]:=cpt DONE)
  DONE(*refait_table_clients*);

  (*Les operations possibles sur les comptes*)
  CONSTANT nombre_maximum_operations=10,
           ouvrir="OUVRIR", deposer="DEPOSER",
           consulter="CONSULTER", retirer="RETIRER";
  Boolean FUNCTOR
    mutation(string VALUE client; positif VALUE somme)
    TABLE mutable VALUE actes=mutable(nombre_maximum_operations);

  Boolean FUNCTION operation
    (string VALUE op,nom,prenom,numero; positif VALUE somme)
  DECLARE(*operation*)
    string VALUE client="["+nom+","_+prenom+":"_+numero+"]"
  DO(*operation*)
    print(line,"***",op,_"compte"_,client,_"montant"_,somme,"Fr.***")
  TAKE
    IF actes ENTRY op THEN
      actes[op][client,somme]
    DEFAULT
      print(line,___"###operation inconnue###")
    TAKE FALSE DONE
  DONE(*operation*);

  Boolean FUNCTION verifier
    (string VALUE client)
  DECLARE(*verifier*)
    Boolean VALUE ok=
      IF
        clients.entree(client)
      THEN
        clients[client]~=NIL
      DEFAULT FALSE DONE
  DO(*verifier*)
    UNLESS ok THEN
      print(line,___"###compte inexistant###")
```

```
credit_suisse                                  Vax Newton Compiler 0.2c1
Page    9
                    Source listing

        DONE
      TAKE ok DONE(*verifier*)
    BEGIN(*comptes*)
      actes[ouvrir]:=
        BODY
          mutation(string VALUE client; positif VALUE somme)
        DO
          IF clients.quantite>rempli_max*clients.capacite THEN
            refait_table_clients(clients.quantite/rempli_min)
          DONE
        TAKE
          IF TAKE
            IF clients.entree(client) THEN
              clients[client]~=NIL
            DEFAULT FALSE DONE
          THEN
            print(line,___"###compte existe deja###")
          TAKE FALSE ELSE

          IF somme<depot_minimum THEN
            clients[client]:=NIL;
            print(line,___"###depot initial"_,edit(somme,2,0),
                       "Fr. insuffisant###")
          TAKE FALSE

          DEFAULT
            clients[client]:=compte(somme);
            print(line,___"compte ouvert, credit initial:"_,
                       somme,"Fr.")
          TAKE TRUE DONE
        DONE(*ouvrir*);
      actes[deposer]:=
        BODY
          mutation(string VALUE client; positif VALUE somme)
        DO TAKE
          UNLESS verifier(client) THEN
            FALSE
          DEFAULT
            print(line,___,somme,"Fr. deposes; nouveau credit:"_,
                       clients[client].deposer(somme).credit,"Fr.")
          TAKE TRUE DONE
        DONE(*deposer*);
      actes[consulter]:=
        BODY
          mutation(string VALUE client; positif VALUE somme)
        DO TAKE
          UNLESS verifier(client) THEN
            FALSE
          DEFAULT
            print(line,___"credit actuel:"_,clients[client].credit,"Fr.",
                  line,___"retrait de"_,somme,"Fr."_);
            IF clients[client].credit-somme>=depot_minimum THEN
```

```
                print("possible")
              DEFAULT
                print("impossible; montant disponible"_,
                      clients[client].credit-depot_minimum,"Fr.")
              DONE
            TAKE TRUE DONE
          DONE(*consulter*);
        actes[retirer]:=
          BODY
            mutation(string VALUE client; positif VALUE somme)
          DO TAKE
            UNLESS verifier(client) THEN
              FALSE ELSE
            IF clients[client].retirer(somme)=somme THEN
              print(line,___"retrait de"_,somme,"Fr. effectue",
                    line,___"solde:"_,clients[client].credit,"Fr.")
            TAKE TRUE

            DEFAULT
              print(line,___"###retrait de"_,somme,"Fr. impossible###",
                    line,___"credit"_,clients[client].credit,"Fr. insuffisant")
            TAKE FALSE DONE
          DONE(*retirer*)
      INNER
        print(page,"*****Etat final des comptes*****");
        clients.parcourir
          (BODY
             action(string VALUE client; compte REFERENCE cpte)
           DO
             print(line,client,column(40));
             CONNECT cpte THEN
               print(credit,"Fr.")
             DEFAULT print("###compte vide###") DONE
           DONE);
        print(line,"<<<FIN>>>")
      END(*comptes*);
      /* /*EJECT*/ */
```

Le module *comptes* ne possède pas d'attribut, mais une fonction d'indiçage *opération.* Il a la structure particulière suivante.

```
module identificateur
   index identificateur
declare
   suite_de_déclarations
begin
   suite_d_énoncés
inner
   suite_d_énoncés
end
```

Un tel module possède un prologue, comportant la partie déclarative et la suite d'énoncés placés entre **begin** et **inner** ainsi qu'un épilogue comportant la suite d'énoncés placée entre **inner** et **end**. Le prologue du module est exécuté à l'entrée du bloc contenant sa déclaration; l'épilogue n'est exécuté qu'au moment de quitter ce bloc. C'est au moyen de l'épilogue du module *comptes* qu'est imprimé l'état final des comptes. On notera que dans le cas d'une coroutine, il n'est pas possible de séparer, par ce moyen, la partie exécutable en un prologue et un épilogue.

L'ensemble des comptes est implanté au moyen de la variable *clients*; au début, il y est stocké une table de capacité *taille_initiale = 15*, construite au moyen de la fonction *table_hachée*. Dès que le taux de remplissage de cette table dépasse *rempli_max = . 8*, il est fait usage de la procédure *refait_table_clients* pour remplacer la table *clients* par une autre, de capacité plus grande: avec la technique d'implantation retenue ici, il n'est pas recommandé de remplir complètement une table (le temps d'insertion des dernières composantes peut devenir prohibitif); la valeur *rempli_max* a été choisie de manière heuristique. On remarque que ces tables agrandies sont produites par la fonction *table_câblée*; le fonctionnement correct du programme montre que les deux représentations sont bien équivalents et qu'elle peuvent coexister sans problèmes.
Ces différentes opérations sur les comptes ont été implantées au moyen de la table *actes* sous la forme d'objets procéduraux du type foncteur *mutation*. Les composantes de cette table sont indicées par le nom de l'opération correspondante; elles y sont insérées lors de l'exécution du prologue du module.

```
credit_suisse                                   Vax Newton Compiler 0.2c1
Page   11
                    Source listing

 string VARIABLE nom,prenom,operation,numero_compte;
 positif VARIABLE montant;

 CONSTANT
   lettres={FROM "A" TO "Z",FROM "a" TO "z"},
   lettres_tiret_soul={FROM "A" TO "Z",FROM "a" TO "z","-","_"},
   chiffres={#0123456789#},
   chiffres_soul={#0123456789#,"_"};

 string VARIABLE ligne; Boolean VARIABLE ok;

 string EXPRESSION cherche_mot=
 (*Cherche, dans  ligne , prochain mot de la forme:
     mot = lettre[{lettre|"-"|"_"}lettre]
   Stocke FALSE en  ok  si un tel mot n'est pas trouve.
   Elimine de  ligne  le mot retenu
 *)
   DECLARE
     string VALUE mot=lettres_tiret_soul SPAN (ligne:=" "-ligne)
   DO
     ligne:=lettres_tiret_soul-ligne;
     ok:=ok/\lettres STARTS mot/\lettres ENDS mot
   TAKE mot DONE;

 string EXPRESSION cherche_nombre=
 (*Cherche, dans  ligne , prochain nombre de la forme:
     nombre = chiffre[{chiffre|"_"}chiffre]
   Stocke FALSE en  ok  si un tel nombre n'est pas trouve.
   Resulte dans le nombre duquel les "_" ont ete supprimes.
   Elimine de  ligne  le nombre retenu
 *)
   DECLARE
     string VARIABLE nombre:=chiffres_soul SPAN (ligne:=" "-ligne)
   DO
     ligne:=chiffres_soul-ligne;
     ok:=ok/\chiffres STARTS nombre/\chiffres ENDS nombre;
     WHILE "_" IN nombre REPEAT
       nombre:=nombre LEFTOF "_"+nombre LEFTCUT "_"
     REPETITION
   TAKE nombre DONE;

 positif EXPRESSION cherche_positif=
 (*Cherche, dans  ligne , prochain nombre et resulte dans la valeur
   entiere positive associee.
   Stocke FALSE en  ok  si le nombre retenu est nul ou superieur a
   integer MAX .
   Elimine de  ligne  le nombre retenu
 *)
   DECLARE
     string VARIABLE nombre:=cherche_nombre;
     integer VARIABLE n:=0,dig
   DO
```

```
credit_suisse                                    Vax Newton Compiler 0.2c1
Page   12
                    Source listing

          THROUGH nombre VALUE d REPEAT
            IF
              ok:=ok/\n<=(integer MAX-(dig:=ORD d-ORD "0"))%10
            THEN n:=10*n+dig DONE
          REPETITION;
          ok:=ok/\n>0
        TAKE n MAX 1 DONE;

      integer VARIABLE num:=0
    DO(*credit_suisse*)
      print("*****Mutations sur les comptes bancaires*****",line);
      UNTIL end_file REPEAT
        read(ligne);
        print(line,edit((num:=SUCC num),4,0),"--->",ligne);
        ok:=TRUE;
        operation:=UPCASE cherche_mot;
        nom:=cherche_mot; prenom:=cherche_mot;
        numero_compte:=cherche_nombre; montant:=cherche_positif;
        IF ok/\" "-ligne="" THEN
          (*La commande satisfait a la syntaxe requise*)
          IF
            comptes[operation,nom,prenom,numero_compte,montant]
          THEN
            print(line,edit(num,4,0),"--->mutation effectuee")
          DEFAULT
            print(line,edit(num,4,0),"--->mutation non effectuee")
          DONE
        DEFAULT
          print(line,edit(num,4,0),"--->commande incorrecte")
        DONE;
        line
      REPETITION;
      print(line,"*****Fin des mutations*****")
    DONE(*credit_suisse*)

      ****  No messages were issued  ****
```

Le reste du programme est de conception relativement simple; au moyen des primitives de traitement de texte vues au chapitre 6, il n'est pas très difficile de décortiquer les commandes et de vérifier leur bienfacture avant de les faire exécuter.

Dans la fonction *cherche_positif*, on peut relever la manière dont on s'y prend pour s'assurer que l'expression *10 * n + dig* ne dépasse pas *integer* **max**; il ne faut évidemment pas utiliser l'expression *10 * n + dig <= integer* **max**: il est par contre facile de vérifier que l'expression *n<= (integer* **max** *- dig) % 10* est équivalente et qu'elle ne peut provoquer de dépassement de capacité.

Remarque:

Il est clair que les considérations développées dans ce chapitre peuvent être étendues à la réalisation de tables indicées par d'autres types de données que les chaînes. L'essentiel est de pouvoir définir une fonction de hachage efficace qui applique l'ensemble des indices possibles sur l'intervalle approprié des valeurs entières.

Chapitre 9

Réalisation d'un interprète d'expressions arithmétiques

Le dernier exemple du chapitre précédent est une sorte d'interprète de commandes. En se basant sur une démarche analogue, il est possible de réaliser un interprète d'expressions arithmétique: chaque commande sera une expression que l'interprète devra évaluer. Les commandes que cet interprète acceptera satisferont à la syntaxe suivante:

commande = assignation;
assignation = [variable *:=*] expression;
variable = lettre {alphanumérique};
alphanumérique = lettre |chiffre| _;
lettre = *A* |*B* | ... *Z* | *a* | *b* | ...*z*;
chiffre = *0* | *1* | ... *9*;
expression = formule {optop formule};
optop = **min** | **max**
formule = [addop] terme {addop terme};
addop = + | -;
terme = facteur {mulop facteur};
mulop = * | /;
facteur = application [^ facteur];
application = opérande | foncop application;
foncop = **abs** | **ln** | **sin** | **cos** | **tg** | **asin** | **acos** | **atg**;
opérande = variable | entier | *(*assignation*);*
entier = chiffre {chiffre}.

Convention:

Les opérateurs en caractères gras seront représentés en encadrant le mot correspondant des symboles ` (accent grave) et ' (accent aigu ou apostrophe).

Exemple d'un fichier de commandes:

```
a
a:=760
qq:=(a+98)/(un_identificateur:=65*2-66/4)
230
48/100
(ici:=78)/
  457-ici
cet_identificateur_est_vraiment_tres_long:=987654321
1/cet_identificateur_est_vraiment_tres_long
zero:=0
  H2SO4 :=  a *  ici / 314 - ( Truc := 198/( 23* ici + 7  ) / a  ^ 3 / qq -2 )
mille:=1000
mille+9
moins_un:=-1
un:=`abs'moins_un
moins_un`max'mille`min'un
moins_un`max'-mille`min'un
-mille`max'un`min'+mille
-mille`max'moins_un`min'+mille
truc:=-Truc`min'Truc
pi_:= `atg' 1 * 4
moins_pi_ := - `atg' 1  * 4
e^(-e)
`ln'e
`ln'2/`ln'10
e^e^e
e2:=e^2
e^(-2)-1/e2
millard:=10^9
millardieme:=1/millard
`ln'millard/`ln'10
`ln'millardieme/`ln'10
maxint:=2^31-1
minint:=-2^31+1
137
23*8-22*6
00987-00456
164/7-(89-67/2^3^2+(56/(8*77)-147)*71-9)-(88-76/5)*6^2
presque_pi:=355/113
pi-presque_pi
pi-pi_
pi_-presque_pi
pi`max'presque_pi
pi`min'presque_pi
`sin'(pi/3)
`asin'(1/2)
`acos'(1/2)
`asin'(t:=-3^(1/2)/2)
`acos't
`tg'(pi/4)
12345679*63
ln2:=`ln'(deux:=2)
ceci:=ici/cet_identificateur_est_vraiment_tres_long*(a:=45)^ln2
qq+87
qq+(pp:=67)/98-a*(mille+1/mille)/(mille^2-98)/(q:=ici/(2-ici)-(r:=56))-1
pi2:=pi^2
p:=34+a:=78
v:=(99/98)
w:=(v-10^6*p/ici-un_identificateur)/ici
q
qqq
```

```
qqq-89
ici-78/5)
yy:=(ici-74/96)/mille
ici+q*23
j:=(i:=98765)
uuu:=87/78
trois:=3
quatre:=4
(trois^2+quatre^2)^(1/2)
3^3^3
(3^3)^3
tu:=(-675)^3`min'675^3
tu:=
(23-784
`qq'666
trois`opt'quatre`min'cinq
gaga:=gaga+1
ici:=ici+2
srahlec:=7
(pinar:=5)
dd:=8#4-797
dd:=8+4-797
a:=(79-(h:=a+un_identificateur)-[89-ici)/5-truc)/2
a:=(79-h-(89-ici)/5-truc)/2
Srahlec_Pinar:=srahlec+pinar+(Srahlec:=1/srahlec)+(Pinar:=1/pinar)
AaBb_1234:=1234
AABB_1234:=AaBb_1234*AaBb_1234
grand:=mille^9
pi^2/10
```

Chacune des lignes de ce fichier est une commande (correcte ou incorrecte); ces commandes seront interprétées séquentiellement dans l'ordre indiqué.

Le résultat de leur interprétation est donné ci-après; ces résultats appellent les commentaires suivants:

- Des variables peuvent être introduites dynamiquement à tout moment; initialement, la valeur d'une variable n'est pas définie sauf pour *pi* et *e* qui sont initialisées par les constantes mathématiques correpondantes.

- Dans un nom de variable, les lettres majuscules et minuscules homologues sont considérées comme distinctes.

- Une fois initialisée une variable conserve sa valeur pour les commandes suivantes de la même session.

- A la fin d'une session, il est donné une liste des valeurs de toutes les variables créées lors de cette dernière.
- Chaque commande correcte est suivie de la valeur résultant de son interprétation.

- Les calculs sont faits en arithmétique réelle, bien qu'il n'ait été prévu que des notations pour les constantes entières.

- Les commandes incorrectes donnent lieu à un (voire parfois plusieurs) message(s) d'erreur; ce message est suivi d'une indication de la partie de la commande au début de laquelle l'erreur a été décelée (cette partie peut être vide).

```
***Interprete d'expressions arithmetiques***

a
   Resultat:  ###Nombre inconnu ou inexistant###

a:=760
   Resultat:  760.0000000000

qq:=(a+98)/(un_identificateur:=65*2-66/4)
   Resultat:  7.559471365639

230
   Resultat:  230.0000000000

48/100
   Resultat:  .4800000000000

(ici:=78)/
   ###ERREUR: operande mal forme ou manquant###
      decelee au debut de---><---
   Resultat:  ###Nombre inconnu ou inexistant###

  457-ici
   Resultat:  379.0000000000

cet_identificateur_est_vraiment_tres_long:=987654321
   Resultat:  987654321.0000

1/cet_identificateur_est_vraiment_tres_long
   Resultat:  +.10124999999873437&-08

zero:=0
   Resultat:  .0

  H2SO4 :=  a *  ici / 314 - ( Truc := 198/( 23* ici + 7  ) / a  ^ 3 / qq -2 )
   Resultat: 190.7898089172

mille:=1000
   Resultat: 1000.000000000

mille+9
   Resultat:  1009.000000000

moins_un:=-1
   Resultat: -1.000000000000

un:=`abs'moins_un
   Resultat:  1.000000000000

moins_un`max'mille`min'un
   Resultat:  1.000000000000

moins_un`max'-mille`min'un
   Resultat: -1.000000000000

-mille`max'un`min'+mille
   Resultat:  1.000000000000

-mille`max'moins_un`min'+mille
   Resultat: -1.000000000000

truc:=-Truc`min'Truc
   Resultat: -1.999999999967
```

```
pi_:= `atg' 1 * 4
   Resultat:  3.141592653590

moins_pi_ := - `atg' 1  * 4
   Resultat: -3.141592653590

e^(-e)
   Resultat:  +.65988035845312535&-01

`ln'e
   Resultat:  1.000000000000

`ln'2/`ln'10
   Resultat:  .3010299956640

e^e^e
   Resultat:  3814279.104760

e2:=e^2
   Resultat:  7.389056098931

e^(-2)-1/e2
   Resultat:  .0

millard:=10^9
   Resultat:  1000000000.000

millardieme:=1/millard
   Resultat:  +.10000000000000000&-08

`ln'millard/`ln'10
   Resultat:  9.000000000000

`ln'millardieme/`ln'10
   Resultat: -9.000000000000

maxint:=2^31-1
   Resultat:  2147483647.000

minint:=-2^31+1
   Resultat: -2147483647.000

137
   Resultat:  137.0000000000

23*8-22*6
   Resultat:  52.00000000000

00987-00456
   Resultat:  531.0000000000

164/7-(89-67/2^3^2+(56/(8*77)-147)*71-9)-(88-76/5)*6^2
   Resultat:  7753.304885349

presque_pi:=355/113
   Resultat:  3.141592920354

pi-presque_pi
   Resultat:  -.26676418907189969&-06

pi-pi_
   Resultat:  .0
```

```
pi_-presque_pi
   Resultat:   -.26676418907189969&-06

pi`max'presque_pi
   Resultat:   3.141592920354

pi`min'presque_pi
   Resultat:   3.141592653590

`sin'(pi/3)
   Resultat:   .8660254037844

`asin'(1/2)
   Resultat:   .5235987755983

`acos'(1/2)
   Resultat:   1.047197551197

`asin'(t:=-3^(1/2)/2)
   Resultat:  -1.047197551197

`acos't
   Resultat:   2.617993877991

`tg'(pi/4)
   Resultat:   1.000000000000

12345679*63
   Resultat:   777777777.0000

ln2:=`ln'(deux:=2)
   Resultat:   .6931471805599

ceci:=ici/cet_identificateur_est_vraiment_tres_long*(a:=45)^ln2
   Resultat:   +.11051194623179026&-05

qq+87
   Resultat:   94.55947136564

qq+(pp:=67)/98-a*(mille+1/mille)/(mille^2-98)/(q:=ici/(2-ici)-(r:=56))-1
   Resultat:   7.243934022524

pi2:=pi^2
   Resultat:   9.869604401089

p:=34+a:=78
   ###Ligne incompletement traitee###
     reste--->:=78<---
   Resultat:   79.00000000000

v:=(99/98)
   Resultat:   1.010204081633

w:=(v-10^6*p/ici-un_identificateur)/ici
   Resultat:  -12986.32054636

q
   Resultat:  -57.02631578947

qqq
   Resultat:   ###Nombre inconnu ou inexistant###

qqq-89
   Resultat:   ###Nombre inconnu ou inexistant###
```

```
ici-78/5)
   ###Ligne incompletement traitee###
      reste--->)<---
   Resultat:  62.40000000000

yy:=(ici-74/96)/mille
   Resultat:  +.77229166666666666&-01

ici+q*23
   Resultat: -1233.605263158

j:=(i:=98765)
   Resultat:  98765.00000000

uuu:=87/78
   Resultat:  1.115384615385

trois:=3
   Resultat:  3.000000000000

quatre:=4
   Resultat:  4.000000000000

(trois^2+quatre^2)^(1/2)
   Resultat:  5.000000000000

3^3^3
   Resultat:  +.76255974849869999&+13

(3^3)^3
   Resultat:  19683.00000000

tu:=(-675)^3`min'675^3
   Resultat: -307546875.0000

tu:=
   ###ERREUR: operande mal forme ou manquant###
      decelee au debut de---><---
   Resultat:  ###Nombre inconnu ou inexistant###

(23-784
   ###ERREUR: ) attendue###
      decelee au debut de---><---
   Resultat:  ###Nombre inconnu ou inexistant###

`qq'666
   ###ERREUR: operande mal forme ou manquant###
      decelee au debut de--->`qq'666<---
   Resultat:  ###Nombre inconnu ou inexistant###

trois`opt'quatre`min'cinq
   ###Ligne incompletement traitee###
      reste--->`opt'quatre`min'cinq<---
   Resultat:  3.000000000000

gaga:=gaga+1
   Resultat:  ###Nombre inconnu ou inexistant###

ici:=ici+2
   Resultat:  80.00000000000

srahlec:=7
   Resultat:  7.000000000000
```

```
(pinar:=5)
   Resultat:  5.000000000000

dd:=8#4-797
   ###Ligne incompletement traitee###
      reste--->#4-797<---
   Resultat:  8.000000000000

dd:=8+4-797
   Resultat: -785.0000000000

a:=(79-(h:=a+un_identificateur)-[89-ici)/5-truc)/2
   ###ERREUR: operande mal forme ou manquant###
      decelee au debut de--->[89-ici)/5-truc)/2<---
   ###ERREUR: ) attendue###
      decelee au debut de---><---
   Resultat:  ###Nombre inconnu ou inexistant###

a:=(79-h-(89-ici)/5-truc)/2
   Resultat: -39.65000000002

Srahlec_Pinar:=srahlec+pinar+(Srahlec:=1/srahlec)+(Pinar:=1/pinar)
   Resultat:  12.34285714286

AaBb_1234:=1234
   Resultat:  1234.000000000

AABB_1234:=AaBb_1234*AaBb_1234
   Resultat:  1522756.000000

grand:=mille^9
   Resultat:  +.10000000000000000&+28

pi^2/10
   Resultat:  .9869604401089
<<<FIN>>>
```

- En général, une commande incorrecte ne donne pas lieu à un résultat numérique; il peut cependant en être produit un si la partie de la commande qui précède l'erreur a elle-même la forme d'une commande correcte.

- Pour le reste, les opérateurs ont leur signification usuelle; on notera que l'opérateur ^ dénote l'exponentiation.

```
***Valeurs finales des variables***

trois:  3.000000000000
ceci:  +.11051194623179026&-05
un_identificateur:  113.5000000000
e:  2.718281828459
quatre:  4.000000000000
millard:  1000000000.000
h:  158.5000000000
a: -39.65000000002
j:  98765.00000000
maxint:  2147483647.000
AaBb_1234:  1234.000000000
w: -12986.32054636
truc: -1.999999999967
millardieme:  +.10000000000000000&-08
gaga:  ###Nombre inconnu ou inexistant###
t: -.8660254037844
presque_pi:  3.141592920354
yy:  +.77229166666666666&-01
qqq:  ###Nombre inconnu ou inexistant###
tu:  ###Nombre inconnu ou inexistant###
Pinar:  .2000000000000
moins_pi_: -3.141592653590
mille:  1000.000000000
e2:  7.389056098931
Truc: -1.999999999967
srahlec:  7.000000000000
grand:  +.10000000000000000&+28
i:  98765.00000000
pi2:  9.869604401089
ln2:  .6931471805599
H2SO4:  190.7898089172
v:  1.010204081633
un:  1.000000000000
qq:  7.559471365639
deux:  2.000000000000
pi_:  3.141592653590
ici:  80.00000000000
pi:  3.141592653590
minint: -2147483647.000
uuu:  1.115384615385
cet_identificateur_est_vraiment_tres_long:  987654321.0000
pp:  67.00000000000
Srahlec:  .1428571428571
moins_un: -1.000000000000
Srahlec_Pinar:  12.34285714286
p:  79.00000000000
q: -57.02631578947
pinar:  5.000000000000
AABB_1234:  1522756.000000
r:  56.00000000000
dd: -785.0000000000
zero:  .0
<<<FIN>>>
```

La structure de l'interprète *calculette* est donnée dans la figure 19.

program *calculette*

constant *non_défini*
procedure *imprime*

real **functor** *monadique*

monadique **value** *ident, neg, abso, log,*
sinus, cosinus, tan, arc_sinus, arc_cosinus, arc_tan

module *monad_table* **index** *applique_monad*

real **function** *applique_monad*

real **functor** *dyadique*

dyadique **value** *mini, maxi, plus, moins,*
fois, divd, puiss

module *dyad_table* **index** *applique_dyad*

real **function** *applique_dyad*

module *var* **index** *entrée*

real **access** *entrée*

constant *lettre, chiffre, optop, addop, mulop, foncop*

alphabet **value** *alpha_num*

real **function** *assign*

Figure 19

Ainsi, on peut subdiviser la partie déclarative du programme *calculette* en cinq parties principales.

La partie exécutable du programme lit les commandes; il les fait analyser et évaluer au moyen de la fonction *assign* située à la fin de la partie déclarative.

La fonction *assign* fait appel au module *var* pour la gestion des variables, au module *dyad_table* pour l'évaluation des opérations dyadiques ainsi qu'au module *monad_table* pour celle des opérations monadiques.

Les valeurs réelles sont imprimées, selon un format approprié, au moyen de la procédure *imprime* .

Le gros du travail d'analyse est réalisé au moyen de la fonction *assign*; sa structure est donnée dans la figure 20.

real **function** *assign*

procedure *erreur*
real **function** *entier*
real **function** ~~*opérande*~~ assign
real **function** *application*
real **function** *opérande*
real **function** *facteur*
real **function** *terme*
real **function** *formule*
real **function** *express*

Figure 20

```
calculette                                          Vax Newton Compiler 0.2c2
Page    1
                        Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]CALCULETTE.NEW*/ */
     1 PROGRAM calculette DECLARE
     4   CONSTANT non_defini=-INFINITY;
    10
    10   PROCEDURE imprime(real VALUE x) DO
    18   (*Imprime, selon disposition adequate, la valeur  x *)
    18     UNLESS FINITE x THEN
    22       print(_"###Nombre inconnu ou inexistant###") ELSE
    28     IF x=0 THEN
    33       print(_".0") ELSE
    39     IF ABS x>integer MAX\/ABS x<.1 THEN
    52       print(x)
    56     DEFAULT edit(x,15,12-FLOOR(ln(ABS x)/ln(10))) DONE
    80   DONE(*imprime*);
    82   /* /*EJECT*/ */
```

La partie initiale du programme *calculette* ne nécessite pas de longs commentaires. Dans tout le système, il a été pris la convention de produire la valeur *non_défini* = - **infinity** pour chaque opération dont le résultat n'est mathématiquement pas défini.

Les deux parties suivantes du programme traitent les opérations monadiques (à un opérande) et respectivement les opérations dyadiques (à deux opérandes). Leurs structures sont semblables; on y voit l'usage d'une table de foncteurs, indicée par les noms des opérateurs concernés, pour implanter leur sémantique par l'intermédiaire des fonctions d'indiçage respectivement *applique_monad* et *applique_dyad*. On y remarque que ces fonctions propagent la valeur *non_défini* dès qu'un de leurs opérandes n'a pas de valeur définie.

```
calculette                                        Vax Newton Compiler 0.2c2
Page      2
                   Source listing

   real FUNCTOR monadique(real VALUE x)
     VALUE(*operateurs monadiques predefinis*)
     ident=        BODY monadique DO TAKE x DONE,
     neg=          BODY monadique DO TAKE -x DONE,
     abso=         BODY monadique DO TAKE ABS x DONE,
     log=          BODY monadique DO TAKE
                     IF x>0 THEN
                       ln(x)
                     DEFAULT non_defini DONE
                   DONE,
     sinus=        BODY monadique DO TAKE sin(x) DONE,
     cosinus=      BODY monadique DO TAKE cos(x) DONE,
     tan=          BODY monadique DO TAKE
                     sin(x)/cos(x)
                   DONE,
     arc_sinus=    BODY monadique DO TAKE
                     IF ABS x<=1 THEN
                       arctan(x/sqrt(1-x**2))
                     DEFAULT non_defini DONE
                   DONE,
     arc_cosinus=BODY monadique DO TAKE
                     IF ABS x<=1 THEN
                       pi/2-arctan(x/sqrt(1-x**2))
                     DEFAULT non_defini DONE
                   DONE,
     arc_tan=      BODY monadique DO TAKE
                     arctan(x)
                   DONE;

 MODULE monad_table  INDEX applique_monad  DECLARE
 (*La table des operateurs monadiques predefinis*)
   CONSTANT max_monad=20;

   monadique TABLE m_table VALUE monad=m_table(max_monad);

   real FUNCTION applique_monad
     (string VALUE m; real VALUE x)
   (*Applique l'operateur  m  a la valeur x *)
   DO(*applique_monad*)TAKE
     IF FINITE x THEN
       monad[m][x]
     DEFAULT x DONE
   DONE(*applique_monad*)
 DO(*monad_table*)
   monad["+"]:=ident; monad["-"]:=neg; monad["`abs'"]:=abso;
   monad["`ln'"]:=log;
   monad["`sin'"]:=sinus; monad["`cos'"]:=cosinus; monad["`tg'"]:=tan;
   monad["`asin'"]:=arc_sinus; monad["`acos'"]:=arc_cosinus;
   monad["`atg'"]:=arc_tan
 DONE(*monad_table*);
 /* /*EJECT*/ */
```

```
calculette                                   Vax Newton Compiler 0.2c2
Page    3
                  Source listing

  real FUNCTOR dyadique(real VALUE x,y)
    VALUE(*operateurs dyadiques predefinis*)
    mini= BODY dyadique DO TAKE x MIN y DONE,
    maxi= BODY dyadique DO TAKE x MAX y DONE,
    plus= BODY dyadique DO TAKE x+y DONE,
    moins=BODY dyadique DO TAKE x-y DONE,
    fois= BODY dyadique DO TAKE x*y DONE,
    divd= BODY dyadique DO TAKE
            IF y~=0 THEN
              x/y
            DEFAULT INFINITY DONE
          DONE,
    puiss=BODY dyadique DO TAKE
            IF FRAC y=0 THEN
              x**(FLOOR y) ELSE
            IF x>0 THEN
              exp(y*ln(x))
            DEFAULT non_defini DONE
          DONE;

MODULE dyad_table  INDEX applique_dyad DECLARE
(*La table des operateurs dyadiques predefinis*)
  CONSTANT max_op=20;

  dyadique TABLE dyad_table VALUE dyad=dyad_table(max_op);

  real FUNCTION applique_dyad
    (string VALUE d; real VALUE x,y)
  (*Applique l'operateur  d  au couple  [x,y] *)
  DO TAKE
    UNLESS FINITE x THEN
      x ELSE
    UNLESS FINITE y THEN
      y
    DEFAULT dyad[d][x,y] DONE
  DONE(*applique_dyad*)
DO(*dyad_table*)
  dyad["`min'"]:=mini; dyad["`max'"]:=maxi;
  dyad["+"]:=plus; dyad["-"]:=moins;
  dyad["*"]:=fois; dyad["/"]:=divd; dyad["^"]:=puiss
DONE(*dyad_table*);
/* /*EJECT*/ */
```

Dans une version plus fine du système, il serait nécessaire de reprogrammer certains opérateurs de manière à s'assurer que leur résultat ne sorte pas des bornes imposées par l'ordinateur sur lequel est installé le système.

Exemple:

```
plus =
  body dyadique do take
    case sign x when
    1 then take
       if y > real max - x then
          infinity
       default x + y done
    0 then y
    - 1 then take
       if y < - real max - x then
          - infinity
       default x + y done
    done (* case sign x *)
  done (* body dyadique *)
```

On y voit que des précautions sont nécessaires sur la manière d'organiser les tests; ainsi, si la valeur de x est positive, il ne faut évidemment pas tester l'expression $x + y >$ *real* **max**: par contre la condition mathématiquement équivalente $y >$ *real* **max** - x ne peut produire de débordement.

La quatrième partie du programme est une table extensible *var* de variables indicées par les noms de ces dernières. C'est l'épilogue du module *var* qui fait imprimer les contenus de toutes les variables à la fin de l'exécution du système.

calculette Vax Newton Compiler 0.2c2
Page 4
Source listing

```
613 MODULE var  INDEX entree  DECLARE
618 (*Une memoire extensible de variables reperees
618   par des identificateurs
618 *)
618   CONSTANT taille_init=20,rempli_min=.5,rempli_max=.8;
633
633   real TABLE r_table VARIABLE t:=r_table(taille_init);
644
644   real ACCESS entree
647     (string VALUE id)
652   (*Resulte en un repere a la variable  id ; si cette variable
652     n'existe pas, la cree et l'initialise a  non_defini .
652   *)
652   DO(*entree*)
653     UNLESS t ENTRY id THEN
658       UNLESS CARD t<rempli_max*CAPACITY t THEN
667         THROUGH
668           (r_table(CARD t%rempli_min)=:t) INDEX id VALUE rid
683         REPEAT
684           t[id]:=rid
690         REPETITION
691       DONE;
693       t[id]:=non_defini
699     DONE
700   TAKE t[id] DONE(*entree*)
706 BEGIN(*var*)
707   t["pi"]:=pi; t["e"]:=e
720 INNER(*var*)
721   page; print("***Valeurs finales des variables***"); line;
730   THROUGH t INDEX id  VALUE nid  REPEAT
737     line; print(id,":"_); imprime(nid)
751   REPETITION;
753   line; print("<<<FIN>>>")
759 END(*var*);
761 /* /*EJECT*/ */
```

En dernière partie, la fonction *assign* incorpore des algorithmes d'analyse syntaxique. En paramètre, il lui est transmis une variable contenant la commande à interpréter.

```
calculette                                    Vax Newton Compiler 0.2c2
Page    5
                  Source listing

CONSTANT lettre={FROM "A" TO "Z",FROM "a" TO "z"},
         chiffre={FROM "0" TO "9"},
         optop="`min'`max'",
         addop={"+","-"},
         mulop={"*","/"},
         foncop="`abs'`ln'`sin'`cos'`tg'`asin'`acos'`atg'";
alphabet VALUE
  alpha_num=lettre+chiffre+"_";

real FUNCTION assign
  (string REFERENCE objet)
 (*Traite: assign=[variable:=]express *)
DECLARE(*assign*)
 PROCEDURE erreur(string VALUE mess)DO
   print(line,___"###ERREUR:"_,mess,"###",
         line,______"decelee au debut de--->",objet,"<---");
   objet:=""
 DONE(*erreur*);

 real FUNCTION entier
   (string VALUE ent)
 (*Traite: entier=chiffre{chiffre} *)
 DECLARE real VARIABLE x:=0 DO
   THROUGH ent VALUE d REPEAT
     x:=10*x+(ORD d-ORD "0")
   REPETITION
 TAKE x DONE;

 real FUNCTION operande
 (*Traite: operande=variable|entier|"("assign")" *)
 DECLARE real VARIABLE f:=non_defini DO
   IF lettre STARTS (objet:=" "-objet) THEN
     f:=var[alpha_num SPAN objet];
     objet:=alpha_num-objet ELSE
   IF chiffre STARTS objet THEN
     f:=entier(chiffre SPAN objet);
     objet:=chiffre-objet ELSE
   IF "(" STARTS objet THEN
     objet:=objet LEFTCUT "(";
     f:=assign(objet);
     IF ")" STARTS " "-objet THEN
       objet:=objet LEFTCUT ")"
     DEFAULT erreur(") attendue"); f:=non_defini DONE
   DEFAULT erreur("operande mal forme ou manquant") DONE
 TAKE f DONE(*operande*);
 /* /*EJECT*/ */
```

```
calculette                                        Vax Newton Compiler 0.2c2
Page    6
                    Source listing

real FUNCTION application
(*Traite: application=operande|foncop application *)
DECLARE
  string VALUE f=" "-objet TOLEFT "'"
DO(*application*)TAKE
  UNLESS "`" STARTS f/\"'" ENDS f/\f IN foncop THEN
    operande
  DEFAULT
    monad_table[f, (objet:=objet LEFTCUT "'"; application)]
  DONE
DONE(*application*);

real FUNCTION facteur
(*Traite: facteur=application["^"facteur] *)
DECLARE
  real VALUE a=application
DO(*facteur*)TAKE
  IF "^" STARTS " "-objet THEN
    dyad_table["^", a,(objet:=objet LEFTCUT "^"; facteur)]
  DEFAULT a DONE
DONE(*facteur*);

real FUNCTION terme
(*Traite: terme=facteur{mulop facteur} *)
DECLARE real VARIABLE t:=facteur DO
  WHILE mulop STARTS " "-objet REPEAT
    t:=dyad_table[objet LEFTOCC mulop,
                  t,(objet:=objet LEFTCUT mulop; facteur)]
  REPETITION
TAKE t DONE(*terme*);

real FUNCTION formule
(*Traite: formule=[addop]terme{addop terme} *)
DECLARE
  real VARIABLE f:=
    IF addop STARTS (objet:=" "-objet) THEN
      monad_table[objet LEFTOCC addop,
                  (objet:=objet LEFTCUT addop; terme)]
    DEFAULT terme DONE
DO
  WHILE addop STARTS " "-objet REPEAT
    f:=dyad_table[objet LEFTOCC addop,
                  f,(objet:=objet LEFTCUT addop; terme)]
  REPETITION
TAKE f DONE(*formule*);
/* /*EJECT*/ */
```

```
    real FUNCTION express
    (*Traite: express=formule{optop formule} *)
    DECLARE real VARIABLE e:=formule; string VARIABLE op DO
      WHILE
        "`" STARTS (op:=" "-objet TOLEFT "'")/\"'" ENDS op/\op IN optop
      REPEAT
        e:=dyad_table[op, e,(objet:=objet LEFTCUT "'"; formule)]
      REPETITION
    TAKE e DONE(*express*)
  DO(*assign*)TAKE
    IF TAKE
      IF lettre STARTS (objet:=" "-objet) THEN
        ":=" STARTS " "-(alpha_num-objet)
      DEFAULT FALSE DONE
    THEN
      var[alpha_num SPAN objet]:=(objet:=objet LEFTCUT ":="; express)
    DEFAULT express DONE
  DONE(*assign*);

  string VARIABLE ligne; real VARIABLE resultat
DO(*calculette*)
  print(page,"***Interprete d'expressions arithmetiques***",line);
  UNTIL end_file REPEAT
    read(ligne); print(line,ligne); resultat:=assign(ligne);
    UNLESS ligne-" "="" THEN
      print(line,___"###Ligne incompletement traitee###",
             line,______"reste--->",ligne,"<---")
    DONE;
    line; print(___"Resultat:"_); imprime(resultat);
    line
  REPETITION;
  print("<<<FIN>>>")
DONE(*calculette*)

     ****  No messages were issued  ****
```

Ces algorithmes d'analyse sont basés sur la méthode de la descente récursive. En gros, l'idée consiste à confier à une procédure déterminée l'analyse de chacune des notions non terminales de la grammaire; ces procédures sont construites en se rapprochant, autant que possible, des productions syntaxiques correspondantes. Chacune des procédures analysera, depuis son début, la chaîne *objet* qui lui est confiée; elle en éliminera la partie initiale qu'elle a été à même d'analyser. Au cas où l'une des procédures décèle que la partie initiale de la chaîne *objet* ne peut pas dériver de la notion qu'elle est chargée d'analyser, elle le signale au moyen d'un libellé d'erreur.

Remarque:

Cette méthode de descente récursive ne peut être appliquée à n'importe quelle grammaire; dans le cas présent, elle nécessite d'ailleurs quelques adaptations. Lorsqu'elle est applicable, la méthode de descente récursive présente l'avantage d'être relativement facile à programmer; à l'exécution, elle est efficace dès le moment où elle présuppose qu'il est possible de choisir la dérivation appropriée de chaque notion en fonction du contexte initial de la chaîne à analyser.

On va maintenant examiner les principaux cas rencontrés dans cet interprète. Un cas simple est celui de la procédure *opérande*. On a les trois dérivations possibles:

opérande = variable;
opérande = entier;
opérande = *(*assignation*)*;

Le caractère initial de la chaîne de la chaîne *objet* permet de décider celle des trois dérivations qu'il est nécessaire d'appliquer:

- Si ce caractère est une lettre, il faut dériver "opérande" en "variable"; cette variable est la sous-chaîne *alpha_num* **span** *objet*. On l'élimine d'*objet* en remplaçant le contenu de cette variable par l'autre partie *alpha_num-objet* de la coupure considérée.

- Si ce caractère est un chiffre, il faut dériver "opérande" en "entier"; cet entier est la sous-chaîne *chiffre* **span** *objet*. Ce cas est analogue au précédent.

- Si ce caractère est une parenthèse ouvrante, il faut dériver "opérande" en "*(*assignation*)* ". Après avoir éliminé de la chaîne *objet* sa parenthèse initiale, il faut appliquer (récursivement) la procédure *assignation* pour analyser et éliminer d'*objet* la partie qui dérive de "assignation". A ce stade, *objet* doit débuter par une parenthèse fermante que l'on élimine; il faut signaler une erreur si l'on ne trouve pas cette parenthèse fermante.

- Si ce caractère n'est ni une lettre, ni un chiffre, ni une parenthèse ouvrante, la partie initiale de la chaîne *objet* ne peut dériver de la notion "opérande": il faut signaler une erreur.

On a la formulation de principe suivante:

```
procedure opérande do
   if lettre starts objet then
      objet := alpha_num-objet else
   if chiffre starts objet then
      objet := chiffre-objet else
   if "(" starts objet then
      objet := objet leftcut "("; assignation;
      if ")" starts objet then
         objet := objet leftcut ")"
      default erreur (") attendue ") done
   default erreur ("opérande mal formé ou manquant ") done
done (* opérande *)
```

Sous cette forme, on n'a qu'un algorithme d'analyse pur; cet algorithme ne fait qu'analyser la chaîne considérée *objet* et signaler les erreurs de sysntaxe éventuelle.

En règle générale, un tel algorithme doit être complété en fonction de l'application que l'on est en train de traiter; ce traitement implique normalement des actions sur les parties de la chaîne que l'on a réussi à analyser: avant d'éliminer ces dernières, il faut donc leur appliquer les actions sémantiques appropriées.

Dans le cas présent, l'action sémantique associée a une notion telle que "opérande" consiste a obtenir la valeur de l'opérande correspondant. Il est donc tout naturel de remplacer la procédure *opérande* par une fonction de type *real* dont le résultat sera la valeur de l'opérande que cette fonction a pu analyser:

```
real function opérande declare
   real variable f := non_défini
do (* opérande *)
   if lettre starts objet then
      f := var [alpha_num span objet];
      objet := alpha_num-objet else
   if chiffre starts objet then
      f := entier (chiffre span objet);
      objet := chiffre-objet else
   if "(" starts objet then
      objet := objet leftcut "(";
      f := assign (objet);
      if ")" starts objet then
         objet := objet leftcut ")"
      default
         erreur (") attendue"); f := non-défini
      done
   default erreur ("opérande mal formé au manquant") done
take f done (* opérande *)
```

Dans le cas d'une variable, l'action sémantique appliquée à la chaîne *alpha_num* **span** *objet* consiste à en obtenir la valeur au moyen du module *var* et de la stocker dans la variable *f* introduite pour retourner le résultat de la fonction *opérande*.

Dans le cas d'un entier, il est fait appel à une fonction spécifique *entier* pour convertir la chaîne *chiffre* **span** *objet* dans la valeur correspondante du type *real*.

Finalement, dans le cas d'une assignation parenthésée, en lieu et place d'une procédure sans paramètre *assignation*, l'action sémantique appropriée résulte de application récursive de la fontion *assign* à la chaîne *objet* (de laquelle la parenthèse intiale a été enlevée).

On en arrive ainsi à une formulation proche de celle du programme *calculette*. On a cependant admis que des espaces blancs peuvent figurer entre les différents éléments syntaxiques des commandes. Les espaces éventuels sont éliminés, aux endroits appropriés des algorithmes d'analyse, en faisant usage de l'expression " " - *objet*. Comme deuxième exemple d'analyse, on va réaliser la procédure *facteur*; on rappelle que la production correspondante a la forme suivante:

facteur = application [^ facteur];

Cette production doit être comprise comme une abréviation des deux productions suivantes:

facteur = application;
facteur = application ^ facteur;

La deuxième de ces productions est récursive à droite. A priori, le caractère initial de la chaîne *objet* ne permet pas de décider laquelle de ces deux productions il faut choisir. Dans le cas présent, la chose n'est pas trop gênante; en-effet, les deux productions débutent par le même non-terminal "application". Il est donc toujours possible de débuter l'analyse par celle de cette dernière notion; on suppose qu'une procédure *application* ad-hoc le fera et éliminera la partie correspondante de la chaîne *objet*. On remarque que le caractère ^ ne peut provenir que d'une application de la règle "facteur = application ^ facteur" (c'est, en-effet, le seul endroit où il apparaît dans la grammaire donnée); il s'ensuit que si la chaîne *objet* résiduelle débute par un ^, c'est cette règle qu'il fallait appliquer: après avoir éliminé ce ^, il suffit alors d'appliquer récursivement la procédure *facteur*. Si, après l'analyse de la notion "assignation", la chaîne résiduelle *objet* ne débute par par un ^, seule la règle courte "facteur = application" est applicable: l'analyse est donc achevée. On aboutit à la procédure suivante:

```
procedure facteur do
   application;
   if "^" starts objet then
      objet := objet leftcut "^";
      facteur
   done
done (*facteur*)
```

Cette procédure a une structure proche de celle de la production syntaxique donnée initialement; en particulier, la règle récursive à droite y correspond à une application récursive de la procédure *facteur*.

Dans un deuxième temps,.il est nécessaire d'introduire les actions sémantiques appropriés. Il faut, pour cela transformer (dans le cas présent) *facteur* en une fonction, de type *real*, sans paramètre; on admet qu'il en a été de même de la procédure *application :*

```
real function facteur declare
   real value a = application
do (* facteur *) take
   if "^" starts objet then
      dyad_table ["^"] [a, (objet := objet leftcut "^"; facteur)]
   default a done
done (* facteur *)
```

On remarque la manière dont l'action sémantique associée à la règle "facteur = application ^ facteur" est réalisée par l'intermédiaire de la table d'opérateur dyadiques *dyad_table*. On peut constater qu'en cas d'usage répété, l'opérateur ^ sera associé de droite à gauche; ceci résulte de la production récursive à droite; le schéma de programmation adopté conserve cette propirété.

Le lecteur peut constater que le cas de la fonction *application* n'est pas très différent; on a les deux règles:

application = opérande;
application = foncop application;

Un opérande ne peut débuter que par une lettre, un chiffre ou une parenthèse gauche; par contre, la notion "foncop" dénote un mot encadré des symboles ` et '. Le contexte initial de la chaîne *objet* permet donc de savoir celle des deux dérivations qu'il est nécessaire d'appliquer. De nouveau, la règle récursive à droite sera transcrite au moyen d'une application récursive de la fonction *application*. Pour décider si la chaîne *objet* débute par un mot de l'ensemble "foncop", on a défini la chaîne *foncop* formée de la concaténation des mots possibles, encadrées de leurs délimiteurs. Soit *f* la chaîne *objet* **toleft** " ' "; l'expression " `" **starts** *f* / \ *f* **in** *foncop* est vraie ssi *objet* débute par un tel mot.

On va maintenant considérer le cas de la procédure *terme*; on part de la dérivation donnée.

terme = facteur {mulop facteur};

Il s'agit d'une abréviation des deux règles suivantes:

terme = facteur;
terme = terme mulop facteur;

Cette fois-ci, on a une règle récursive à gauche, ce qui pose un problème nouveau. Le contexte initial de la chaîne *objet* ne permet pas de décider celle des deux règles qu'il faut appliquer; d'autre part, il faudra faire en sorte que les opérateurs de l'ensemble *mulop* soient associés de gauche à droite (conséquence de la règle récursive à gauche). Une dérivation quelconque de la notion "terme" résulte de *n* applications successives de la règle récursive, pour une valeur

entière *n* non négative arbitraire, suivie d'une application de la règle courte "terme = facteur". Ceci donne lieu à un arbre syntaxique de la forme donnée à la figure 21.

Figure 21

La dérivation débute par un facteur; celui-ci est suivi de *n* occurences du groupe "mulop facteur" qu'il faut associer de proche en proche. Ceci peut être fait au moyen d'un schéma itératif. Pour cela, on commence par analyser le facteur initial au moyen d'une application de la procédure *facteur*; on considère que ce facteur est un terme (règle courte). On remarque qu'un caractère de l'ensemble *mulop* ne peut résulter que de l'application de la règle récursive "terme = terme mulop facteur"; par conséquent tant que le reste de la chaîne *objet* débute par un tel caractère, le terme que l'on a récolté constitue le début du membre droit de cette règle: il faut continuer par l'analyse du groupe "mulop facteur". On arrive donc à la procédure suivante:

```
procedure terme do
   facteur;
   while mulop start objet repeat
      objet := objet  leftcut mulop;
      facteur
   repetition
done (* terme *)
```

L'insertion des productions sémantiques appropriées découle immédiatement de la discussion précédente:

```
real function terme declare
   real variable t := facteur
do (* terme *)
   while mulop starts objet repeat
      t := dyad_table [objet leftocc mulop]
            [t, (objet := objet leftcut mulop; facteur)]
   repetition
take t done (* terme *)
```

On voit le rôle de la variable *t* dans laquelle sont récoltés, de proche en proche, les termes (partiels) déjà évalués.
La conception des fonctions *formules* et *express* est analogue, avec la légère complication supplémentaire, dans la première, qu'une formule peut débuter par un opérateur optionnel de l'ensemble *addop*.

L'analyse de la notion "assignation" est plus délicate; on part donc de la production:

assignation = [variable :=] expression;

Cette production est une abréviation des deux règles suivantes:

assignation = variable := expression;
assignation = expression;

Il est facile de voir que "expression" peut donner lieu à une chaîne débutant par "variable"; il s'ensuit que le contexte initial de la chaîne *objet* ne permet pas toujours de décider la règle à appliquer. Dans le cas de la grammaire proposée, une variable débute par une lettre; si *objet* commence par une lette, on considère la sous-chaîne *alpha_num-objet* (c'est-à-dire *objet* de laquelle on a éliminé sa variable initiale); si cette sous-chaîne débute par :=, il faut analyser *objet* ou moyen de la règle longue "assignation = variable := expression". Dans tous les autres cas, on applique la règle courte "assignation = expression". La procédure *application* a donc la forme suivante:

```
procedure assignation do
   if take
      if lettre starts objet then
         ":=" starts alpha_num-objet
      default false done
   then
      objet := objet leftcut ":="
   done;
   expression
done (* assignation *)
```

La structure de cette procédure est relativement proche de celle de la dérivation donnée en premier lieu. Cependant, lorsque l'on veut ajouter les productions sémantiques, il est plus simple de séparer complètement les deux cas. De plus, au lieu d'une fonction *assignation* sans paramètre, le traitement requis sera directement incorporé à la partie exécutable de la fonction d'analyse principale *assign*. Cette dernière fonction a donc la forme:

```
real function assign
   (string reference objet)
declare
   / * ... cf figure 20 ... * /
do (* assign *) take
   if take
      if lettre starts objet then
         ":=" starts alpha_num-objet
      default  false  done
   then
      var [alpha_num span objet] :=
         (objet := objet leftcut ":="; express)
   default express done
done (* assign *)
```

Chapitre 10

Piles, queues et listes

Une liste est une suite linéaire d'éléments; dans le cas général, on admet que des éléments peuvent être insérés en un point quelconque d'une liste et qu'un élément arbitraire peut en être retiré.

Au chapitre 2, on a vu, dans le programme *listes*, un exemple de liste triée. Une liste triée est une liste dont les éléments successifs sont ordonnés par ordre croissant, ou selon toute autre relation d'ordre définie au préalable.

Une pile est une liste dans laquelle l'insertion et l'élimination d'éléments ont toujours lieu à la même extrémité. Il s'ensuit que l'élément retiré en premier d'une pile est celui qui y a été inséré le plus récemment (last-in, first-out ou LIFO en anglais). L'extrémité privilégiée d'une pile, c'est-à-dire l'élément qui y a été inséré le plus récemment en est le sommet (figure 22).

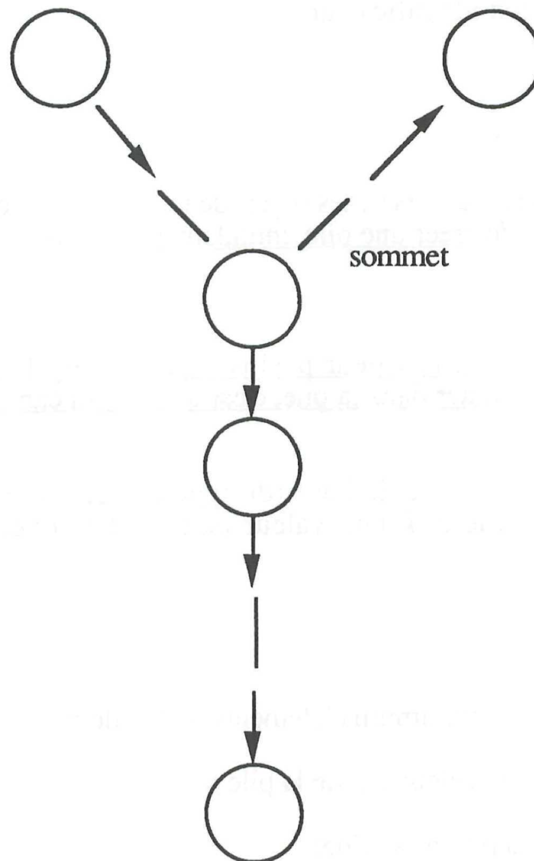


Figure 22

Une queue est une liste dans laquelle l'insertion d'éléments nouveaux a lieu à l'une des extrémités et l'élimination d'éléments a lieu à l'autre extrémité. Il s'ensuit que l'élément retiré en premier est celui qui y a été inséré en premier (first-in, first-out on FIFO en anglais). L'avant d'une queue est un élément qui y a été inséré en premier ; c'est donc l'élément situé à l'avant d'une queue qui en est retiré en premier. L'arrière d'une queue est l'élément qui y a été inséré en dernier; c'est l'élément qui en sera retiré en dernier (Figure 23).

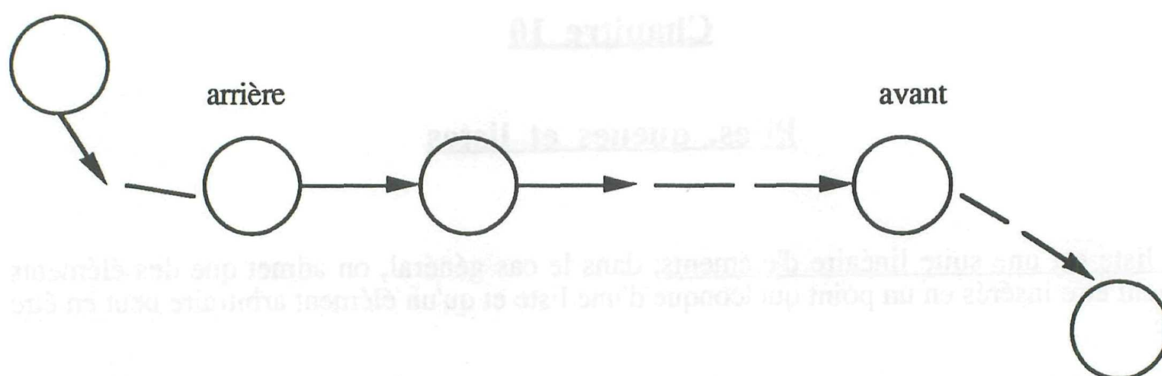


Figure 23

Vu leur usage relativement fréquent, le langage Newton permet de déclarer et utiliser des piles et des queues de capacité bornée de manière analogue à ce qui est le cas pour les rangées et les tables. En particulier une pile ou une queue est considérée comme un objet.

Un type pile est déclaré de la manière suivante:

indication_de_type **stack** identificateur

Exemple:

character **stack** *char_pile*

Cette déclaration définit le type *char_pile*; les objets de ce type seront des piles de caractères. Un générateur de pile permet de créer une pile, initialement vide, d'un type pile spécifique:

type_pile *(*expression_entière*)*

L'expression entière doit livrer une valeur positive qui sera égale au nombre maximum d'éléments susceptibles de coexister dans la pile, c'est-à-dire à sa capacité.

Soient *p* une pile, *b* une valeur du type de base (du type des éléments) de cette pile, *v* une variable du type de base de la pile et *k* une valeur entière. Les opérations suivantes sont disponibles.

Interrogateurs:

capacity *p*	Le nombre maximum d'éléments de la pile *p*.
card *p*	Le nombre d'éléments de la pile *p*
empty *p*	Vrai ssi la pile *p* est vide: **empty** *p* **==** **card** *p* = *0*
full *p*	Vrai ssi la pile *p* est pleine: **full p == card** *p* **= capacity** *p*

Constructeur:

p **push** *b*	Introduit, dans la pile *p*, un nouvel élément de valeur *b*; utilisée comme expression, cette opération livre la pile concernée *p*. **Condition d'emploi: ~ full** *p*.

Sélecteurs:

p **top** — La valeur de l'élément inséré le plus récemment dans la pile *p*, c'est-à dire son sommet.
Condition d'emploi: ~ empty *p*.

p [k] — L'élément situé au niveau *k* de la pile *p*; pour cette opération, on considère que le sommet d'une pile est au niveau *0*, l'élément en dessous du sommet au niveau *-1* et ainsi de suite.
Condition d'emploi:
- card *p* < *k* /\ *k* <= *0*

Destructeur:

v **pop** *p* — Elimine de la pile *p* l'élément placé en son sommet et en stocke la valeur dans la variable *v*; utilisée comme expression, cette opération livre pour résultat la pile concernée *p*.
Condition d'emploi: ~ empty *p*.

Itérateur:

through *p*
  index *k*
  value *b*
repeat
  suite_d_énoncés
repetition

Parcourt élément par élément depuis le fonds jusqu'au sommet, la pile *p*. Pour chaque élément, effectuera la suite_d_énoncés placée entre **repeat** et **repetition**; dans cette suite *b* dénote la valeur de l'élément considéré et *k* son niveau. Les clauses **index** *k* et **value** *b* sont facultatives. Utilisé comme expression, cet itérateur livre comme résultat la pile concernée *p*.

Comparaisons:

Deux piles *p* et *q* du même type peuvent être comparées au moyen des relations *p* = *q* et *p* ~ = *q*; il s'agit d'une comparaison d'objet: les deux piles seront égales ssi il s'agit de la même pile.

Une manipulation de queues est semblable; un type queue est déclaré de la manière suivante:

indication_de_type **queue** identificateur

Une queue , initialement vide, est créée au moyen d'un générateur de queue de la forme:

type_queue *(*expression_entière*)*

De nouveau, l'expression entière livre pour valeur la capacité de la queue: cette valeur sera donc positive.

Soient *q* une queue , *b* une valeur de son type de base, *v* une variable de son type de base et *k* une valeur entière. Les opérations suivantes sont disponibles:

Interrogateurs:

capacity *q* — Le nombre maximum de composantes de la queue *q*.

card *q* — Le nombre d'éléments de la queue *q*.

empty *q* — Vrai ssi la queue *q* est vide:
    empty *q* == **card** *q* = *0*

full *q* — Vrai ssi la queue *q* est pleine:
    full *q* == **card** *q* = **capacity** *q*

Constructeur:

q **append** *b* — Appond à la queue *q* un nouvel élément de valeur *b*. Utilisée comme expression, cette opération livre pour résultat la queue concernée *q*.
Condition d'emploi: ~ full *q*

Sélecteurs:

q **front** — La valeur de l'élément à l'avant de la queue *q* (donc de l'élément que y a été inséré en premier).
Condition d'emploi: ~ empty *q*

q **back** — La valeur de l'élément à l'arrière de la queue *q* (donc de l'élément qui y a été inséré en dernier lieu)
Condition d'emploi: ~ empty *q*

q [k] — La valeur de l'élément de rang *k* de la queue *q*; pour cette opération, l'élément à l'avant de la queue a le rang *1*, le suivant le rang 2 et ainsi de suite jusqu'à l'élément à l'arrière de la queue dont le rang égale **card** *q*.
Condition d'emploi: *1* <= *k* /\ *k* <= **card** *q*

Destructeur:

v **from** *q* — Elimine de la queue *q* l'élément qui y a été inséré en premier et stocke sa valeur dans la variable *v*; utilisée comme expression, cette opération livre la queue concernée *q*.
Condition d'emploi: ~ empty *q*.

Itérateur:

through *q*
  index *k*
  value *b*
repeat
  suite_d_énoncés
repetition

Parcourt, élément par élément de l'avant vers l'arrière, la queue *q*. Pour chaque élément effectue la suite d'énoncés entre **repeat** et **repetition**; dans cette suite, *b* dénote la valeur de l'élément et *k* son rang. Les clauses **index** *k* et **value** *b* sont facultatives. Utilisé comme expression, cet itérateur livre pour résultat la queue *q*.

Comparaisons:

Les comparaisons d'égalité *q = r* et d'inégalité *q~ =r* sont possibles entre deux queues *q* et *r* du même type. Elles seront considérées comme égales ssi il s'agit de la même queue.

Remarques:

- Utilisés comme expressions, les constructeurs *p* **push** *b* et *q* **append** *b* ainsi que les destructeurs *v* **pop** *p* et *v* **from** *q* doivent le plus souvent être parenthésés. Les opérations sont du même très bas niveau de priorité que les opérations d'assignation et de dénotation (:=, =:, :=: et →).

- Les sélecteurs postfixés *p* **top**, *q* **front** et *q* **back** sont du même haut niveau de priorité que les opérations d'indiçage: ils n'ont en général pas besoin d'être parenthésés.

On va illustrer piles et queues, ainsi que leurs techniques d'implantation, au moyen de deux programmes.

Dans le premier, *pilistes* , on va montrer comment construire de manière efficace une liste triée en faisant usage d'une pile de listes partielles que l'on fusionne au moment approprié.

La construction d'une liste triée, par insertion successive de ses éléments, est en-effet une opération relativement coûteuse. L'insertion d'un élément exige, en moyenne, le parcours de la moitié de la liste; il s'ensuit que la construction d'une liste triée de *n* éléments prend, en moyenne, un temps proportionnel à *n* ** 2.

Pour autant que la liste soit susceptible d'être établie d'un seul coup, une liste triée de *n* éléments peut être construite en un temps proportionnel à *n* * *ln(n)*. On tient compte pour cela que la fusion de deux listes triées en une seule prend un temps proportionnel à la somme de leurs longueurs.

L'idée de l'algorithme est la suivante. On admet que l'on a une pile de listes *p*. Initialement, on placera en *p* une liste vide; cette liste ne sera jamais manipulée: elle servira de sentinelle pour les opérations suivantes.

Pour chaque valeur *x* à insérer dans la liste triée, on effectue les opérations suivantes:

- On construit une liste triée *lt* contenant le seul élément *x*.

- Tant que la longueur de la liste *lt* est égale à celle de la liste en sommet de *p*, on dépile cette dernière et on la fusionne dans *lt* (c'est-là qu'on se rend compte qu'il est utile d'avoir une sentinelle de longueur nulle au fonds de *p*).

- On empile sur *p* la liste *lt* ainsi obtenue.

On termine l'algorithme en dépilant de *p* les listes qui y sont enregistrées et en les fusionnant, de proche en proche, en une seule liste triée. Cette dernière est le résultat cherché.

La structure générale du programme *pilistes* est donnée dans la figure 24 suivante.

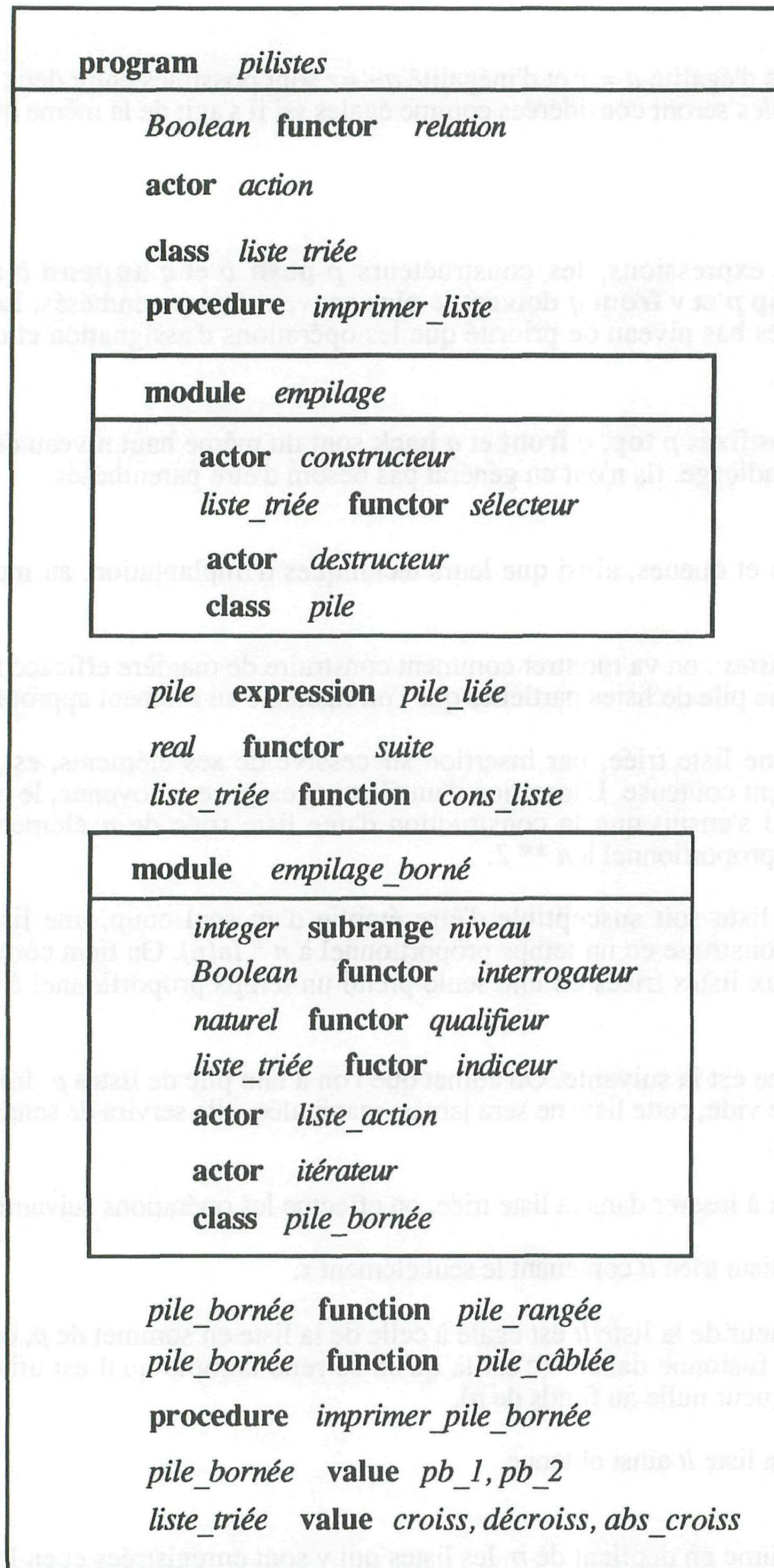

program pilistes
Boolean functor relation
actor action
class liste_triée
procedure imprimer_liste
module empilage
actor constructeur
liste_triée functor sélecteur
actor destructeur
class pile
pile expression pile_liée
real functor suite
liste_triée function cons_liste
module empilage_borné
integer subrange niveau
Boolean functor interrogateur
naturel functor qualifieur
liste_triée fuctor indiceur
actor liste_action
actor itérateur
class pile_bornée
pile_bornée function pile_rangée
pile_bornée function pile_câblée
procedure imprimer_pile_bornée
pile_bornée value pb_1, pb_2
liste_triée value croiss, décroiss, abs_croiss

Figure 24

```
pilistes                                  Vax Newton Compiler 0.2c6
                                    Page    1
                Source listing

/* /*OLDSOURCE=USER2:[RAPIN]PILISTES.NEW*/ */
PROGRAM pilistes DECLARE
  integer SUBRANGE naturel(naturel>=0)
          SUBRANGE positif(positif>0);

  Boolean FUNCTOR relation(real VALUE gauche,droite)VALUE
    inf=BODY relation DO TAKE gauche<droite DONE,
    sup=BODY relation DO TAKE gauche>droite DONE,
    abs_inf=BODY relation DO TAKE ABS gauche<ABS droite DONE;

  ACTOR action(integer VALUE rang; real VALUE val);

  CLASS liste_triee
    VALUE moi
    ATTRIBUTE inferieur,longueur,mettre,premier,enlever,epouser,parcourir
    (relation VALUE inferieur)
  (*Un objet de type  liste_triee  est une liste, triee selon la relation
    inferieur , initialement vide de valeurs reelles.

    Conditions d'emploi:
      inferieur[x,y] NAND inferieur[y,x]
      inferieur[x,y] NOR inferieur[y,x] |>
        (inferieur[y,z]==inferieur[x,z])
      inferieur[x,y]/\inferieur[y,z] |> inferieur[x,z]
  *)
  DECLARE(*liste_triee*)
    (***Representation interne***)
    OBJECT liste(real VALUE val; liste VARIABLE suiv)
                 VARIABLE tete:=NIL;
    integer VARIABLE long VALUE longueur:=0;
    (*Le nombre d'elements de la liste*)

    (***Implantation des operations realisables sur la liste***)
    liste_triee FUNCTION mettre
      (real VALUE x)
    (*Insere dans la liste un nouvel element de valeur  x ;
      le resultat est la liste concernee  moi *)
    DECLARE liste REFERENCE ins->tete DO
      CYCLE cherche_place REPEAT
        CONNECT ins THEN
          UNLESS
            inferieur[val,x]
          EXIT cherche_place DONE;

          ins->suiv
        DEFAULT(*ins=NIL*)
          EXIT cherche_place
        DONE
      REPETITION;
      ins:=liste(x,ins);
      long:=SUCC long
    TAKE moi DONE(*mettre*);
```

```
real EXPRESSION premier=
(*La valeur du premier element de la liste:

  Condition d'emploi:  longueur~=0
*)
  CONNECT tete THEN
    val
  DEFAULT
    print(line,"###premier non applicable a liste vide###")
  RETURN 0 DONE;

liste_triee FUNCTION enlever
  (real REFERENCE xx)
(*Retire de la liste concernee son premier element et en
  stocke la valeur dans la variable reperee par  xx . Resulte
  dans la liste concernee  moi .

  Condition d'emploi:  longueur~=0
*)
DO(*enlever*)
  CONNECT tete THEN
    xx:=val; tete:=suiv; long:=PRED long
  DEFAULT
    print(line,"###enlever non applicable a liste vide###")
  RETURN DONE
TAKE moi DONE(*enlever*);

liste_triee FUNCTION epouser
  (liste_triee VALUE toi)
(*Unit la liste  toi  a la liste concernee  moi ; le resultat
  est la liste fusionnee  moi . Sans effet si  toi=NIL  ou si
  toi=moi ; dans le cas contraire, la liste  toi  sera videe.

  Condition d'emploi:  inferieur=toi.inferieur
*)
DECLARE liste REFERENCE joint->tete DO
  UNLESS toi=NIL\/toi=moi THEN
    UNLESS inferieur=toi.inferieur THEN
      print(line,"###epouser non applicable a listes ordonnees"_
                 "selon relations d'ordre differentes###")
    RETURN DONE;
    (*Dans un premier temps, contruit en  tete  la liste
      fusionnee
    *)
    UNTIL toi.tete=NIL REPEAT
      IF TAKE
        CONNECT joint THEN
          inferieur[toi.tete.val,val]
        DEFAULT TRUE DONE
      THEN
        (*Le premier element de la liste fusionnee vient de
```

```
pilistes                                              Vax Newton Compiler 0.2c6
                                         Page     3
                Source listing

                        toi.tete
                      *)
                      joint:=:toi.tete
                    DONE;
                    (*Le premier element de la fusion est maintenant issu
                      de la variable reperee par  joint .
                    *)
                    joint->joint.suiv
                  REPETITION;
                  (*La fusion est faite; arrange les resultats*)
                  long:=long+(0=:toi.long)
                DONE(*UNLESS toi=NIL\/toi=moi*)
              TAKE moi DONE;

              liste_triee FUNCTION parcourir
                (action VALUE acte)
              (*Parcourt la liste concernee element par element. Pour chaque
                element, effectue l'enonce acte[rang,val]  ou  rang  est le
                rang  et  val  la valeur de l'element. Le resultat est la liste
                concernee  moi .
              *)
              DECLARE
                liste VARIABLE curseur:=tete;
                integer VARIABLE r:=0
              DO
                WITHIN curseur REPEAT
                  acte[(r:=SUCC r),val]; curseur:=suiv
                REPETITION
              TAKE moi DONE
            DO(*liste_triee*)DONE;

            PROCEDURE imprimer_liste(liste_triee VALUE lt) DO
              print(line,"***Impression d'une liste***");
              lt.parcourir
                (BODY action(integer VALUE k; real VALUE elt) DO
                   IF k\5=1 THEN line DONE; edit(elt,15,10)
                 DONE);
              print(line,"<<<FIN>>>")
            DONE(*imprimer_liste*);
            /* /*EJECT*/ */
```

Après quelques déclarations préalables, il apparaît une version modifiée de la classe *liste_triée*; au lieu d'un nom il est fourni comme paramètre des objets de cette classe la relation *inférieur* selon laquelle la liste correspondante sera ordonnée. La fonction *mettre* a été modifiée en conséquence ainsi que la fonction *épouser* qui remplace *marier*; au lieu de fusionner deux listes en une troisième, cette opération joint la liste qui lui est passé en paramètre à celle dont elle est l'attribut: on remarque qu'il faut aussi s'assurer que les listes que l'on fusionne soient ordonnées selon la même relation.

```
pilistes                                        Vax Newton Compiler 0.2c6
                                       Page   4
                    Source listing

MODULE empilage
  ATTRIBUTE pile,constructeur,selecteur,destructeur
(*Definit la classe protocole  pile  dont les objets sont des piles
de listes triees.
*)
DECLARE(*empilage*)
  ACTOR constructeur(liste_triee VALUE val);
  liste_triee FUNCTOR selecteur;
  ACTOR destructeur(liste_triee REFERENCE var);

  CLASS pile
    VALUE moi
    ATTRIBUTE vide,profondeur,empiler,sommet,depiler
    (constructeur VALUE cons;
     selecteur VALUE som;
     destructeur VALUE des)
  (*Un objet du type  pile  est une pile, initialement vide, de listes
    triees. L'implanteur devra fournir, au moyen d'objets proceduraux,
    les operations suivantes:

      cons[lt]     : empile la liste  lt
      som EVAL     : resulte en la liste en sommet de pile
      des[lt_var]  : enleve de la pile la liste inseree en dernier lieu;
                     stocke cette liste dans la variable  lt_var
  *)
  DECLARE(*pile*)
    naturel VARIABLE long VALUE profondeur:=0;
    (*Le nombre de composantes de la pile*)

    Boolean EXPRESSION vide=
    (*Vrai ssi la pile est vide*)
      profondeur=0;

    pile FUNCTION empiler
      (liste_triee VALUE val)
    (*Insere dans la pile un nouvel element  val ; le resultat est la
      pile concernee  moi .
    *)
    DO(*empiler*)
      long:=SUCC long; cons[val]
    TAKE moi DONE(*empiler*);

    liste_triee EXPRESSION sommet=
    (*La valeur de l'element insere le plus recemment dans la pile.

      Condition d'emploi:   ~vide
    *)
      (IF profondeur=0 THEN
         print(line,"###sommet de pile vide non defini###")
       RETURN DONE;
      som EVAL);
```

```
      pile FUNCTION depiler
        (liste_triee REFERENCE var)
      (*Elimine de la pile l'element qui y a ete place le plus recemment;
        stocke sa valeur dans la variable reperee par  var ; le resultat
        est la pile concernee  moi .

        Condition d'emploi:  ~vide
      *)
      DO(*depiler*)
        IF profondeur=0 THEN
          print(line,"###depiler non applicable a pile vide###")
        RETURN DONE;
        des[var]; long:=PRED long
      TAKE moi DONE(*depiler*)
    DO(*pile*)DONE
  DO(*empilage*)DONE;

    pile EXPRESSION pile_liee=
    (*Le resultat est une pile, non bornee a priori, d'objets du type
      liste_triee.
    *)
      DECLARE
        OBJECT liste (liste_triee VALUE val; liste VALUE suiv)
          VARIABLE tete:=NIL
      DO TAKE
        pile BODY
               constructeur(liste_triee VALUE lt)
             DO tete:=liste(lt,tete) DONE,
             BODY selecteur DO TAKE tete.val DONE,
             BODY
               destructeur(liste_triee REFERENCE lt_var)
             DO
               CONNECT tete THEN
                 lt_var:=val; tete:=suiv
               DONE
             DONE)
      DONE(*pile_liee*);
  /* /*EJECT*/ */
```

```
pilistes                                         Vax Newton Compiler 0.2c6
                                      Page   6
                  Source listing

  678    real FUNCTOR suite(positif VALUE n);
  687
  687    liste_triee FUNCTION cons_liste
  690      (positif VALUE quantite; suite VALUE s; relation VALUE inf)
  703    (*Le resultat est une liste de  quantite  elements reels
  703      tries selon la relation  inf ; l'element de rang  k  est
  703      obtenu au moyen de l'expression  s[k] .
  703    *)
  703    DECLARE(*cons_liste*)
  704      pile VALUE p=pile_liee.empiler(liste_triee(inf));
  718      liste_triee VARIABLE lt,st;
  724      real VARIABLE x
  727    DO(*cons_liste*)
  728      print(line,"***Construction d'une liste***");
  735      FOR integer VALUE k FROM 1 TO quantite REPEAT
  744        IF k\5=1 THEN line DONE;
  754        edit((x:=s[k]),15,10);
  770        lt:=liste_triee(inf).mettre(x);
  782        WHILE lt.longueur=p.sommet.longueur REPEAT
  793          p.depiler(st); lt.epouser(st)
  806        REPETITION;
  808        p.empiler(lt)
  814      REPETITION;
  816      print(line,"<<<FIN>>>");
  823      p.depiler(lt);
  830      UNTIL p.sommet.longueur=0 REPEAT
  839        p.depiler(st); lt.epouser(st)
  852      REPETITION
  853    TAKE lt DONE(*cons_liste*);
  857    /* /*EJECT*/ */
```

La fonction *cons_liste* réalise la construction d'une liste triée au moyen d'une pile de listes. En paramètre, il est fourni à cette fonction le nombre d'éléments dont la liste sera composée, la valeur des éléments successifs par l'intermédiaire d'un objet foncteur du type *suite* et la relation d'ordre selon laquelle la liste sera ordonnée.

```
MODULE empilage_borne
 ATTRIBUTE pile_bornee,niveau,
           interrogateur,quantifieur,indiceur,
           liste_action,iterateur
DECLARE(*empilage_borne*)
 integer SUBRANGE niveau
   (niveau<=0
    DEFAULT
      print(line,"###niveau de pile ne peut etre positif###")
    TAKE 0);
 Boolean FUNCTOR interrogateur;
 naturel FUNCTOR quantifieur;
 liste_triee FUNCTOR indiceur(niveau VALUE niv);
 ACTOR liste_action(niveau VALUE niv; liste_triee VALUE lt);
 ACTOR iterateur(liste_action VALUE acte);

 CLASS pile_bornee
   VALUE moi
   INDEX element
   ATTRIBUTE capacite,vide,plein,profondeur,
             empiler,sommet,depiler,parcourir,
             pile_induite
   (positif VALUE capacite;
    interrogateur VALUE vd,pln;
    quantifieur VALUE quant;
    constructeur VALUE cons;
    selecteur VALUE som;
    indiceur VALUE ind;
    destructeur VALUE des;
    iterateur VALUE parc)
  (*Un objet du type  pile_bornee  est une pile d'au maximum
    capacite  listes triees. A tout objet de cette classe
    protocole, l'implanteur devra fournir, au moyen d'objets
    proceduraux, les operations suivantes:

      vd EVAL     : vrai ssi la pile est vide.
      pln EVAL    : vrai ssi la pile est pleine.
      quant EVAL  : le nombre de composantes de la pile.
      cons[lt]    : empile la liste  lt .
      som EVAL    : l'element insere en dernier lieu dans la pile.
      ind[niv]    : l'element au niveau  niv  de la pile.
      des[lt_var]: enleve de la pile la liste qui y a ete inseree
                    le plus recemment; stocke cette liste dans la
                    variable  lt_var .
      parc[acte] : effectue l'enonce  acte[niv,elt]  pour chaque
                    element  elt  de la pile ; le parcours a lieu
                    depuis le fonds jusqu'au sommet de la pile:
                    niv  est le niveau de l'element.
  *)
  DECLARE(*pile_bornee*)
   Boolean EXPRESSION
     vide=vd[](*vrai ssi la pile est vide*),
```

```
  plein=pln[](*vrai ssi la pile est pleine*);
naturel EXPRESSION
  profondeur=quant[](*le nombre d'elements de la pile*);

pile VALUE pile_induite=
(*La pile concernee vue en tant que pile generale*)
  pile(cons, som, des);

pile_bornee FUNCTION empiler
  (liste_triee VALUE val)
(*Place sur la pile un nouvel element de valeur  val ;
  le resultat est la pile  moi .

  Condition d'emploi:  ~plein
*)
DO(*empiler*)
  IF pln[] THEN
    print(line,"###Empiler non applicable a pile bornee pleine###")
  RETURN DONE;
  pile_induite.empiler(val)
TAKE moi DONE;

liste_triee EXPRESSION
  sommet=pile_induite.sommet(*L'element en sommet de la pile*);

liste_triee FUNCTION element
  (niveau VALUE niv)
(*L'element au niveau  niv  de la pile concernee.

  Condition d'emploi:  profondeur+niv>0
*)
DO(*element*)
  UNLESS quant[]+niv>0 THEN
    print(line,"###Element de pile inexistant###")
  RETURN DONE
TAKE ind[niv] DONE(*element*);

pile_bornee FUNCTION depiler
  (liste_triee REFERENCE var)
(*Elimine de la pile concernee l'element qui y a ete insere le
  plus recemment; le resultat est la pile concernee  moi .

  Condition d'emploi: ~vide
*)
DO pile_induite.depiler(var) TAKE moi DONE(*depiler*);

pile_bornee FUNCTION parcourir
  (liste_action VALUE acte)
(*Parcourt la pile concernee depuis le fonds jusqu'au sommet;
  pour chaque element effectue l'enonce  acte[niv,val]  dans
  lequel  niv  est le niveau et  val  la valeur de l'element.
  Le resultat est la pile concernee  moi .
```

```
1120        *)
1120        DO parc[acte] TAKE moi DONE(*parcourir*)
1128      DO(*pile_bornee*)DONE
1130 DO(*empilage_borne*)DONE;
1133
1133      pile_bornee FUNCTION pile_rangee
1136        (positif VALUE capacite)
1141      (*Realise une implantation d'une  pile_bornee  au moyen d'une
1141        rangee et d'un compteur.
1141      *)
1141      DECLARE(*pile_rangee*)
1142        (*Representation interne*)
1142        liste_triee ROW vecteur VALUE tab=vecteur(1 TO capacite);
1155        integer VARIABLE compte:=0
1160      DO(*pile_rangee*)TAKE
1162        pile_bornee
1163          (capacite,
1166           BODY interrogateur DO TAKE compte=0 DONE,
1175           BODY interrogateur DO TAKE compte=capacite DONE,
1184           BODY quantifieur DO TAKE compte DONE,
1191           BODY
1192             constructeur(liste_triee VALUE val)
1198           DO tab[(compte:=SUCC compte)]:=val DONE,
1212           BODY selecteur DO TAKE tab[compte] DONE,
1222           BODY
1223             indiceur(niveau VALUE niv)
1229           DO TAKE tab[compte+niv] DONE,
1239           BODY
1240             destructeur(liste_triee REFERENCE var)
1246           DO liste_triee[NIL]=:tab[(PRED compte=:compte)]=:var DONE,
1265           BODY
1266             iterateur(liste_action VALUE acte)
1272           DO
1273             FOR integer VALUE k FROM 1 BY 1 TO compte REPEAT
1284               acte[k-compte,tab[k]]
1295             REPETITION
1296           DONE)
1298      DONE(*pile_rangee*);
1300
1300      pile_bornee FUNCTION pile_cablee
1303        (positif VALUE capacite)
1308      (*Implante un objet du type  pile_bornee  au moyen d'une pile
1308        predefinie.
1308      *)
1308      DECLARE(*pile_cablee*)
1309        (*Representation interne*)
1309        liste_triee STACK pile_listes VALUE p=pile_listes(capacite);
1320
1320        (*Operations predefinies*)
1320        Boolean EXPRESSION vide=EMPTY p,
1327                           plein=FULL p;
1332        naturel EXPRESSION profondeur=CARD p;
```

```
  PROCEDURE poser(liste_triee VALUE lt) DO
    p PUSH lt
  DONE(*poser*);

  liste_triee EXPRESSION sommet=p TOP;

  liste_triee FUNCTION element
    (niveau VALUE niv)
  DO TAKE p[niv] DONE(*element*);

  PROCEDURE deposer(liste_triee REFERENCE var_lt) DO
    var_lt POP p
  DONE(*deposer*);

  PROCEDURE parcourir(liste_action VALUE acte) DO
    THROUGH p
      INDEX niv
      VALUE p_niv
    REPEAT acte[niv,p_niv] REPETITION
  DONE(*parcourir*)
DO(*pile_cablee*)TAKE
  pile_bornee
    (CAPACITY p,
     interrogateur(vide),interrogateur(plein),
     quantifieur(profondeur),
     constructeur(poser),
     selecteur(sommet),indiceur(element),
     destructeur(deposer),
     iterateur(parcourir))
DONE(*pile_cablee*);
/* /*EJECT*/ */
```

Le module *empilage_borné* inclut la classe protocole *pile_bornée* décrivant des piles dont la capacité est bornée à priori. Une pile borneé est un cas particulier d'une pile; cependant les types *pile* et *pile_bornée* sont distincts. Par contre, une pile bornée peut être vue sous la forme d'une pile par l'intermédiaire de l'attribut *pile_induite*. On remarque que les opérations communes aux deux catégories de piles *empiler*, *sommet* et *défiler* ont été confiées à la pile induite; dans le cas du constructeur *empiler*, il est cependant introduit dans la classe *pile_bornée* le test de validité supplémentaire que ce constructeur ne doit pas porter sur une pile pleine. En terme de langage objet, on dira qu'un objet de la classe *pile_bornée* hérite les opérations *empiler*, *sommet* et *dépiler* de l'objet *pile_induite* de la classe pile.
Au moyen de fonctions génératrices, il est fourni deux implantations de piles bornées. Dans la première *pile_rangée* une pile d'au maximum *capacité* éléments est représentée de manière classique par une rangée *tab* de *capacité* composantes et d'une variable entière *compte* de valeur égale au nombre d'éléments de la pile. Dans la programmation du destructeur, on remarque que la case liberée de *tab* est mise à la valeur **nil**: ceci a été fait pour permettre de récupérer, dès que possible, la mémoire occupée par la liste que l'on a dépilé. Cette représentation interne constitue un bon modèle d'implantation des piles prédéfinies du langage Newton; c'est en se basant sur une telle pile *p* que la fonction génératrice *pile_câblée* représente une pile bornée: on y voit donc l'usage des opérations disponibles sur les piles.

```
  PROCEDURE imprimer_pile_bornee
    (pile_bornee VALUE pb)
  (*Impression de la pile de listes  pb *)
  DO(*imprimer_pile_bornee*)
    print(page,"*****Impression d'une pile de listes*****",line);
    print(line,_____"Capacite:"_,edit(pb.capacite,3,0),
          line,_____"Pile vide?"_,pb.vide,
          line,_____"Pile pleine?"_,pb.plein,
          line,_____"Nombre d'elements:"_,edit(pb.profondeur,3,0));
    pb.parcourir
      (BODY
         liste_action(niveau VALUE niv; liste_triee VALUE lt)
       DO
         print(line,"*****Debut niveau",edit(niv,3,0),"*****");
         imprimer_liste(lt);
         print(line,"*****Fin niveau",edit(niv,3,0),"*****",line)
       DONE);
    print(line,"<<<<<FIN>>>>>")
  DONE(*imprimer_pile_bornee*);

  pile_bornee VALUE pb_1=pile_rangee(5),
                    pb_2=pile_cablee(3);

  pile VALUE p_1=pb_1.pile_induite,
             p_2=pb_2.pile_induite,
             p_3=pile_liee;

  suite VALUE dist_triangle=BODY suite DO TAKE random-random DONE;
  liste_triee VALUE
    croiss    =(print("*****Construit trois listes*****",line);
                cons_liste(25,dist_triangle,inf)),
    decroiss  =cons_liste(20,dist_triangle,sup),
    abs_croiss=cons_liste(15,dist_triangle,abs_inf);
  liste_triee VARIABLE lt
DO(*pilistes*)
  imprimer_liste(croiss);
  imprimer_liste(decroiss);
  imprimer_liste(abs_croiss);
  print(page,"*****Construit et empile quelques listes*****",line);
  FOR integer VALUE k FROM 1 TO pb_1.capacite REPEAT
    pb_1.empiler
     (cons_liste
      (8*k,
       BODY
         suite(positif VALUE n)
       DO TAKE sqrt(k*n)*normal DONE,
       CASE k\3 WHEN
       1 THEN inf|
       2 THEN sup|
       DEFAULT abs_inf DONE))
  REPETITION;
  imprimer_pile_bornee(pb_1);
```

```
pilistes                                    Vax Newton Compiler 0.2c6
                                     Page   12
                    Source listing

      (*Transfere ce qu'on peut dans  pb_2 *)
     UNTIL pb_2.plein REPEAT
       p_1.depiler(lt); p_2.empiler(lt)
     REPETITION;
     imprimer_pile_bornee(pb_1);
     imprimer_pile_bornee(pb_2);
      (*Transfere le reste dans  p_3 *)
     UNTIL p_1.vide REPEAT
       p_1.depiler(lt); p_3.empiler(lt)
     REPETITION;
      (*Transfere  pb_2  dans  pb_1 *)
     UNTIL p_2.vide REPEAT
       p_2.depiler(lt); p_1.empiler(lt)
     REPETITION;
      (*Transfere  p_3  dans  pb_1 *)
     UNTIL p_3.vide REPEAT
       p_3.depiler(lt); p_1.empiler(lt)
     REPETITION;
     imprimer_pile_bornee(pb_1);
     print(page,"*****Consomme quelques mariages*****");
     UNTIL p_1.vide REPEAT
       p_1.depiler(lt);
       IF lt.inferieur=inf THEN
         croiss.epouser(lt) ELSE
       IF lt.inferieur=sup THEN
         decroiss.epouser(lt) ELSE
       IF lt.inferieur=abs_inf THEN
         abs_croiss.epouser(lt)
       DONE
     REPETITION;
     imprimer_liste(croiss);
     imprimer_liste(decroiss);
     imprimer_liste(abs_croiss);
     print(line,"<<<<<FIN DE CES MANOEUVRES>>>>>")
 DONE(*pilistes*)

     ****  No messages were issued  ****
```

La partie exécutable du programme réalise différentes opérations sur les listes *croiss*, *décroiss* et *abs_coirss* ainsi que sur les piles de listes *pb_1*, *pb_2*, *p_1*, *p_2* et *p_3*. On remarque que *p_1* et *p_2* dénotent les mêmes piles que *pb_1* et respectivement *pb_2* vues comme piles et non comme piles bornées. Ainsi, une opération héritée telle que *p_1.dépiler(lt)* a le même effet que *pb_1.dépiler(lt)*.

Résultats:

```
*****Construit trois listes*****

***Construction d'une liste***
    -.2132633535    .0841413624    .4095669143   -.0931271814    .6001421605
     .2554223280   -.2792755404    .7674126777   -.8576792393   -.2142087725
     .0987967279   -.3743483739   -.7218849224    .5218314426   -.5419722117
    -.1440340359   -.1426826675    .4589397297    .6891111279   -.3504286353
    -.4754545361    .1542822035    .6864154759    .4655655305   -.4807417477
<<<FIN>>>
***Construction d'une liste***
    -.2117974805   -.8032534810    .4601015131   -.6468233853    .3649430441
    -.0795005185   -.2900562310   -.2639168742    .2734120127    .1107021053
    -.7797834377    .7924815732    .5545202959   -.3872143054    .0991740002
    -.6388355199   -.2125231087    .0184703722    .5346362041    .3370260760
<<<FIN>>>
***Construction d'une liste***
    -.4815842478    .0194465370    .2194530018    .6288825271   -.1517394164
     .6217948892    .2129822667    .4302759747   -.2715661954    .3573504897
    -.6164970899    .0241870386    .3004087424   -.8091895222   -.2816628468
<<<FIN>>>
***Impression d'une liste***
    -.8576792393   -.7218849224   -.5419722117   -.4807417477   -.4754545361
    -.3743483739   -.3504286353   -.2792755404   -.2142087725   -.2132633535
    -.1440340359   -.1426826675   -.0931271814    .0841413624    .0987967279
     .1542822035    .2554223280    .4095669143    .4589397297    .4655655305
     .5218314426    .6001421605    .6864154759    .6891111279    .7674126777
<<<FIN>>>
***Impression d'une liste***
     .7924815732    .5545202959    .5346362041    .4601015131    .3649430441
     .3370260760    .2734120127    .1107021053    .0991740002    .0184703722
    -.0795005185   -.2117974805   -.2125231087   -.2639168742   -.2900562310
    -.3872143054   -.6388355199   -.6468233853   -.7797834377   -.8032534810
<<<FIN>>>
***Impression d'une liste***
     .0194465370    .0241870386   -.1517394164    .2129822667    .2194530018
    -.2715661954   -.2816628468    .3004087424    .3573504897    .4302759747
    -.4815842478   -.6164970899    .6217948892    .6288825271   -.8091895222
<<<FIN>>>
```

```
*****Construit et empile quelques listes*****

***Construction d'une liste***
   -.3159168436   1.5619467495   3.8410892706  -1.5173150567   -.2357386709
    .0148150799   4.2491045626   -.0297210851
<<<FIN>>>
***Construction d'une liste***
   -.1062636699  -1.7724251848   2.8817777151  -2.1534450410   2.7559129674
   -.9662547725   2.3936937933   2.0369388854   4.0864315946  -3.9164391719
  -6.9261301804   -.7408563357   7.0874895792   5.5369460603  -2.3328120437
   -.5964118956
<<<FIN>>>
***Construction d'une liste***
  -3.3398491110   2.1967764525   4.9957285968   1.9812240370   3.3083602297
   -.1911836264   3.3709409427   1.7851993220   8.6769185917   8.4765754704
   4.0042203849  -1.7081936589   5.2683459345   2.1173692261  -2.1214672173
  -4.7354193995  -6.7462573466   -.0919216110   6.6874754603 -17.9065230362
  -9.9946912435   3.7035720779   5.7123107630   4.7320159738
<<<FIN>>>
***Construction d'une liste***
   1.7774858880   2.2204073564  -3.4034703852   3.4525979349  -2.9589314306
   3.5653696489   4.8996826166   6.8003449482   1.8986405073  11.5257125266
   5.6025828400   5.1671918906  -6.2027561489  11.2675714320    .2039760359
  -4.6193085181 -12.1119703104  -4.7762322639  10.5276538221  -8.4345082061
  -7.9110290238  13.6729060862   5.5913913044   5.0772753549   7.1162321667
 -14.7037195327  15.2775924122  11.7197036125 -13.9240764149  -6.3537684433
   -.3857287195  22.7966630190
<<<FIN>>>
***Construction d'une liste***
    .7189655475  -2.2123577765  -2.9695366610   -.0236136140   5.8771252403
   2.6246790290   2.7690166240   6.0789038486   6.1163197310   8.0651320044
  12.9388322334  -6.2654124791   5.2238365180  -7.0103159785  -7.0886793547
   8.9044148441 -17.6935474705  17.6328449184 -12.4838041006 -22.6955568686
  15.1631293736   6.6211028584  15.5248542258 -15.7653211811   1.0500121429
 -12.0641290158  -6.3389462602 -12.1627924797   6.0440624630  -9.1593912600
  15.7437418609  11.2561188620   5.0973071288   8.6996131804  14.6247588628
  11.4266872277 -22.3660512423  -5.8378860379  28.2270621270  -4.5636496019
<<<FIN>>>
```

```
*****Impression d'une pile de listes*****

     Capacite:   5
     Pile vide?  'FALSE'
     Pile pleine?   'TRUE'
     Nombre d'elements:   5
*****Debut niveau -4*****
***Impression d'une liste***
  -1.5173150567   -.3159168436   -.2357386709   -.0297210851    .0148150799
   1.5619467495   3.8410892706   4.2491045626
<<<FIN>>>
*****Fin niveau -4*****

*****Debut niveau -3*****
***Impression d'une liste***
   7.0874895792   5.5369460603   4.0864315946   2.8817777151   2.7559129674
   2.3936937933   2.0369388854   -.1062636699   -.5964118956   -.7408563357
   -.9662547725  -1.7724251848  -2.1534450410  -2.3328120437  -3.9164391719
  -6.9261301804
<<<FIN>>>
*****Fin niveau -3*****

*****Debut niveau -2*****
***Impression d'une liste***
   -.0919216110   -.1911836264  -1.7081936589   1.7851993220   1.9812240370
   2.1173692261  -2.1214672173   2.1967764525   3.3083602297  -3.3398491110
   3.3709409427   3.7035720779   4.0042203849   4.7320159738  -4.7354193995
   4.9957285968   5.2683459345   5.7123107630   6.6874754603  -6.7462573466
   8.4765754704   8.6769185917  -9.9946912435 -17.9065230362
<<<FIN>>>
*****Fin niveau -2*****

*****Debut niveau -1*****
***Impression d'une liste***
 -14.7037195327 -13.9240764149 -12.1119703104  -8.4345082061  -7.9110290238
  -6.3537684433  -6.2027561489  -4.7762322639  -4.6193085181  -3.4034703852
  -2.9589314306   -.3857287195    .2039760359   1.7774858880   1.8986405073
   2.2204073564   3.4525979349   3.5653696489   4.8996826166   5.0772753549
   5.1671918906   5.5913913044   5.6025828400   6.8003449482   7.1162321667
  10.5276538221  11.2675714320  11.5257125266  11.7197036125  13.6729060862
  15.2775924122  22.7966630190
<<<FIN>>>
*****Fin niveau -1*****

*****Debut niveau  0*****
***Impression d'une liste***
  28.2270621270  17.6328449184  15.7437418609  15.5248542258  15.1631293736
  14.6247588628  12.9388322334  11.4266872277  11.2561188620   8.9044148441
   8.6996131804   8.0651320044   6.6211028584   6.1163197310   6.0789038486
   6.0440624630   5.8771252403   5.2238365180   5.0973071288   2.7690166240
   2.6246790290   1.0500121429    .7189655475   -.0236136140  -2.2123577765
  -2.9695366610  -4.5636496019  -5.8378860379  -6.2654124791  -6.3389462602
  -7.0103159785  -7.0886793547  -9.1593912600 -12.0641290158 -12.1627924797
 -12.4838041006 -15.7653211811 -17.6935474705 -22.3660512423 -22.6955568686
<<<FIN>>>
*****Fin niveau  0*****

<<<<<FIN>>>>>
```

```
*****Impression d'une pile de listes*****

    Capacite:   5
    Pile vide?  'FALSE'
    Pile pleine?  'FALSE'
    Nombre d'elements:   2
*****Debut niveau -1*****
***Impression d'une liste***
  -1.5173150567   -.3159168436   -.2357386709   -.0297210851    .0148150799
   1.5619467495   3.8410892706   4.2491045626
<<<FIN>>>
*****Fin niveau -1*****

*****Debut niveau  0*****
***Impression d'une liste***
   7.0874895792   5.5369460603   4.0864315946   2.8817777151   2.7559129674
   2.3936937933   2.0369388854   -.1062636699   -.5964118956   -.7408563357
   -.9662547725  -1.7724251848  -2.1534450410  -2.3328120437  -3.9164391719
  -6.9261301804
<<<FIN>>>
*****Fin niveau  0*****

<<<<<FIN>>>>>


*****Impression d'une pile de listes*****

    Capacite:   3
    Pile vide?  'FALSE'
    Pile pleine?   'TRUE'
    Nombre d'elements:   3
*****Debut niveau -2*****
***Impression d'une liste***
  28.2270621270  17.6328449184  15.7437418609  15.5248542258  15.1631293736
  14.6247588628  12.9388322334  11.4266872277  11.2561188620   8.9044148441
   8.6996131804   8.0651320044   6.6211028584   6.1163197310   6.0789038486
   6.0440624630   5.8771252403   5.2238365180   5.0973071288   2.7690166240
   2.6246790290   1.0500121429    .7189655475   -.0236136140  -2.2123577765
  -2.9695366610  -4.5636496019  -5.8378860379  -6.2654124791  -6.3389462602
  -7.0103159785  -7.0886793547  -9.1593912600 -12.0641290158 -12.1627924797
 -12.4838041006 -15.7653211811 -17.6935474705 -22.3660512423 -22.6955568686
<<<FIN>>>
*****Fin niveau -2*****

*****Debut niveau -1*****
***Impression d'une liste***
 -14.7037195327 -13.9240764149 -12.1119703104  -8.4345082061  -7.9110290238
  -6.3537684433  -6.2027561489  -4.7762322639  -4.6193085181  -3.4034703852
  -2.9589314306   -.3857287195    .2039760359   1.7774858880   1.8986405073
   2.2204073564   3.4525979349   3.5653696489   4.8996826166   5.0772753549
   5.1671918906   5.5913913044   5.6025828400   6.8003449482   7.1162321667
  10.5276538221  11.2675714320  11.5257125266  11.7197036125  13.6729060862
  15.2775924122  22.7966630190
<<<FIN>>>
*****Fin niveau -1*****

*****Debut niveau  0*****
***Impression d'une liste***
   -.0919216110   -.1911836264  -1.7081936589   1.7851993220   1.9812240370
   2.1173692261  -2.1214672173   2.1967764525   3.3083602297  -3.3398491110
   3.3709409427   3.7035720779   4.0042203849   4.7320159738  -4.7354193995
```

```
   4.9957285968   5.2683459345   5.7123107630   6.6874754603  -6.7462573466
   8.4765754704   8.6769185917  -9.9946912435 -17.9065230362
<<<FIN>>>
*****Fin niveau  0*****

<<<<<FIN>>>>>
```

```
*****Impression d'une pile de listes*****

     Capacite:   5
     Pile vide?  'FALSE'
     Pile pleine?   'TRUE'
     Nombre d'elements:   5
*****Debut niveau -4*****
***Impression d'une liste***
   -.0919216110   -.1911836264  -1.7081936589   1.7851993220   1.9812240370
   2.1173692261  -2.1214672173   2.1967764525   3.3083602297  -3.3398491110
   3.3709409427   3.7035720779   4.0042203849   4.7320159738  -4.7354193995
   4.9957285968   5.2683459345   5.7123107630   6.6874754603  -6.7462573466
   8.4765754704   8.6769185917  -9.9946912435 -17.9065230362
<<<FIN>>>
*****Fin niveau -4*****

*****Debut niveau -3*****
***Impression d'une liste***
 -14.7037195327 -13.9240764149 -12.1119703104  -8.4345082061  -7.9110290238
  -6.3537684433  -6.2027561489  -4.7762322639  -4.6193085181  -3.4034703852
  -2.9589314306   -.3857287195    .2039760359   1.7774858880   1.8986405073
   2.2204073564   3.4525979349   3.5653696489   4.8996826166   5.0772753549
   5.1671918906   5.5913913044   5.6025828400   6.8003449482   7.1162321667
  10.5276538221  11.2675714320  11.5257125266  11.7197036125  13.6729060862
  15.2775924122  22.7966630190
<<<FIN>>>
*****Fin niveau -3*****

*****Debut niveau -2*****
***Impression d'une liste***
  28.2270621270  17.6328449184  15.7437418609  15.5248542258  15.1631293736
  14.6247588628  12.9388322334  11.4266872277  11.2561188620   8.9044148441
   8.6996131804   8.0651320044   6.6211028584   6.1163197310   6.0789038486
   6.0440624630   5.8771252403   5.2238365180   5.0973071288   2.7690166240
   2.6246790290   1.0500121429    .7189655475   -.0236136140  -2.2123577765
  -2.9695366610  -4.5636496019  -5.8378860379  -6.2654124791  -6.3389462602
  -7.0103159785  -7.0886793547  -9.1593912600 -12.0641290158 -12.1627924797
 -12.4838041006 -15.7653211811 -17.6935474705 -22.3660512423 -22.6955568686
<<<FIN>>>
*****Fin niveau -2*****

*****Debut niveau -1*****
***Impression d'une liste***
  -1.5173150567   -.3159168436   -.2357386709   -.0297210851    .0148150799
   1.5619467495   3.8410892706   4.2491045626
<<<FIN>>>
*****Fin niveau -1*****

*****Debut niveau  0*****
***Impression d'une liste***
   7.0874895792   5.5369460603   4.0864315946   2.8817777151   2.7559129674
   2.3936937933   2.0369388854   -.1062636699   -.5964118956   -.7408563357
   -.9662547725  -1.7724251848  -2.1534450410  -2.3328120437  -3.9164391719
  -6.9261301804
<<<FIN>>>
*****Fin niveau  0*****

<<<<<FIN>>>>>
```

```
*****Consomme quelques mariages*****
***Impression d'une liste***
 -14.7037195327 -13.9240764149 -12.1119703104  -8.4345082061  -7.9110290238
  -6.3537684433  -6.2027561489  -4.7762322639  -4.6193085181  -3.4034703852
  -2.9589314306  -1.5173150567   -.8576792393   -.7218849224   -.5419722117
   -.4807417477   -.4754545361   -.3857287195   -.3743483739   -.3504286353
   -.3159168436   -.2792755404   -.2357386709   -.2142087725   -.2132633535
   -.1440340359   -.1426826675   -.0931271814   -.0297210851    .0148150799
    .0841413624    .0987967279    .1542822035    .2039760359    .2554223280
    .4095669143    .4589397297    .4655655305    .5218314426    .6001421605
    .6864154759    .6891111279    .7674126777   1.5619467495   1.7774858880
   1.8986405073   2.2204073564   3.4525979349   3.5653696489   3.8410892706
   4.2491045626   4.8996826166   5.0772753549   5.1671918906   5.5913913044
   5.6025828400   6.8003449482   7.1162321667  10.5276538221  11.2675714320
  11.5257125266  11.7197036125  13.6729060862  15.2775924122  22.7966630190
<<<FIN>>>
***Impression d'une liste***
  28.2270621270  17.6328449184  15.7437418609  15.5248542258  15.1631293736
  14.6247588628  12.9388322334  11.4266872277  11.2561188620   8.9044148441
   8.6996131804   8.0651320044   7.0874895792   6.6211028584   6.1163197310
   6.0789038486   6.0440624630   5.8771252403   5.5369460603   5.2238365180
   5.0973071288   4.0864315946   2.8817777151   2.7690166240   2.7559129674
   2.6246790290   2.3936937933   2.0369388854   1.0500121429    .7924815732
    .7189655475    .5545202959    .5346362041    .4601015131    .3649430441
    .3370260760    .2734120127    .1107021053    .0991740002    .0184703722
   -.0236136140   -.0795005185   -.1062636699   -.2117974805   -.2125231087
   -.2639168742   -.2900562310   -.3872143054   -.5964118956   -.6388355199
   -.6468233853   -.7408563357   -.7797834377   -.8032534810   -.9662547725
  -1.7724251848  -2.1534450410  -2.2123577765  -2.3328120437  -2.9695366610
  -3.9164391719  -4.5636496019  -5.8378860379  -6.2654124791  -6.3389462602
  -6.9261301804  -7.0103159785  -7.0886793547  -9.1593912600 -12.0641290158
 -12.1627924797 -12.4838041006 -15.7653211811 -17.6935474705 -22.3660512423
 -22.6955568686
<<<FIN>>>
***Impression d'une liste***
    .0194465370    .0241870386   -.0919216110   -.1517394164   -.1911836264
    .2129822667    .2194530018   -.2715661954   -.2816628468    .3004087424
    .3573504897    .4302759747   -.4815842478   -.6164970899    .6217948892
    .6288825271   -.8091895222  -1.7081936589   1.7851993220   1.9812240370
   2.1173692261  -2.1214672173   2.1967764525   3.3083602297  -3.3398491110
   3.3709409427   3.7035720779   4.0042203849   4.7320159738  -4.7354193995
   4.9957285968   5.2683459345   5.7123107630   6.6874754603  -6.7462573466
   8.4765754704   8.6769185917  -9.9946912435 -17.9065230362
<<<FIN>>>
<<<<<FIN DE CES MANOEUVRES>>>>>
```

Pour illustrer les queues et leur implantation, on va montrer un algorithme de tri d'une queue de valeurs entières non négatives au moyen d'une rangée de baquets; les baquets individuels seront, eux-aussi, représentés au moyen de queues.

Le principe de l'algorithme est le suivant:
Soit *q* la queue que l'on désire trier; soit *base* une base de numérotation (par exemple *base = 10*)
On dispose de *base* baquets initialement vides: *baquets [0]*, *baquets [1]*, ... *baquets [base - 1]*.
On va supposer que la plus grande composante de *q* possède *m* chiffres dans la base considérée *base*. Le tri se déroule alors en *m* passes successives; chaque passe implique une semaille des éléments de *q* dans les baquets suivie d'une récolte du contenu des baquets successifs dans *q*.
Plus spécifiquement, les passes successives portent sur les chiffres successifs des éléments de *q* en commençant par le chiffre le moins significatif. Lors de la semaille, les éléments de *q* sont retirés à un à un. Pour chaque élément *n*, on extrait le chiffre *c* (dans la base considérée *base*) associé à cette passe; la valeur *n* est alors enfilée dans le baquet d'indice *c*. Lors de la récolte, les baquets sont vidés un à un dans la queue *q* dans l'ordre croissant de leurs numéros.

Exemple:

On donne la queue de *20* entiers suivants:

833; 397; 379, 179; 787;
536; 196; 295; 783; 307;
255; 982; 10; 666; 780;
604; 807; 947; 413; 249

On choisit de faire le tri avec *base = 10* baquets; il s'ensuit que trois passes seront nécessaires. Dans la première passe, la semaille a lieu en fonction du dernier chiffre décimal; les baquets auront les contenus suivants:

baquets [0] : *010; 780*
[1] : -
[2] : *982;*
[3] : *833, 783; 413*
[4] : *604*
[5] : *295; 255*
[6] : *536; 196; 666*
[7] : *397; 787; 307; 807; 947*
[8] : -
[9] : *379; 179; 249*

La récolte donne la queue suivante:

010; 780; 982; 833; 783;
413; 604; 295; 255; 536;
196; 666; 397; 787; 307;
807; 947; 379; 179; 249

A ce stade, les valeurs sont ordonnées en fonction de leur dernier chiffre; dans la deuxième passe, la semaille a lieu en fonction de l'avant dernier chiffre; les baquets auront les contenus suivants:

baquets [0] : *604; 307; 807*
[1] : *010; 413*
[2] : ---
[3] : *833; 536*
[4] : *947; 249*
[5] : *255*
[6] : *666*
[7] : *379; 179*
[8] : *780; 982; 783; 787*
[9] : *295; 196; 397*

La deuxième récolte donne la queue suivante:

604; 307; 807; 010; 413;
833; 536; 947; 249; 255;
666; 379; 179; 780; 982;
783; 787; 295; 196; 397

Cette fois-ci, les valeurs sont ordonnées selon l'avant dernier chiffre; celles dont les avant derniers chiffres sont égaux le sont selon leur dernier chiffre. Cette dernière propriété est vraie à cause du travail fait lors de la première passe et à cause de la gestion de *q* et des baquets en queue. Dans la troisième passe, la semaille a lieu en fonction du troisième chiffre

avant la fin, c'est-à-dire dans le cas particulier en fonction du premier chiffre, des éléments de la queue. Le contenu des baquets est alors le suivant:

baquets *[0]* : *010*
[1] : *179; 196*
[2] : *249; 255; 295*
[3] : *307; 379; 397*
[4] : *413*
[5] : *536*
[6] : *604; 666*
[7] : *780; 783; 787*
[8] : *807; 833*
[9] : *947; 982*

Après la troisième récolte on constate que la queue a bel et bien été triée:

10; 179; 196; 249; 255;
295; 307; 379; 397; 413;
536; 604; 666; 780; 783;
787; 807; 833; 947; 982

Il apparaît évident que le temps nécessaire à cet algorithme est proportionnel au produit de la longueur de la queue concernée par le nombre de passes requises. Une augmentation du nombre de baquets pourra (jusqu'à un certain point) diminuer le nombre de passes, et par conséquent le temps de calcul; par contre, elle impliquera une plus grande occupation de la mémoire. Sauf pour un très petit nombre de baquets, l'augmentation de l'efficacité en fonction de ce nombre est cependant lente; le nombre de passes requises est, en-effet, inversement proportionnel au logarithme du nombre de baquets: ainsi, pour diminuer de moitié le nombre de passes (donc le temps nécessaire au tri), il faut élever au carré le nombre de baquets.

Remarque:

- Cet algorithme de tri peut aussi être adapté au tris de fichiers séquentiels. Sous cette optique, des fichiers auxiliaires temporaires tiendront lieu de baquets.

- Des adaptations sont évidemment nécessaires lorsque les données sur lesquelles le tri porte n'a pas la forme d'entiers non négatifs.

La structure générale du programme *baquets* est donnée à la figure 25.

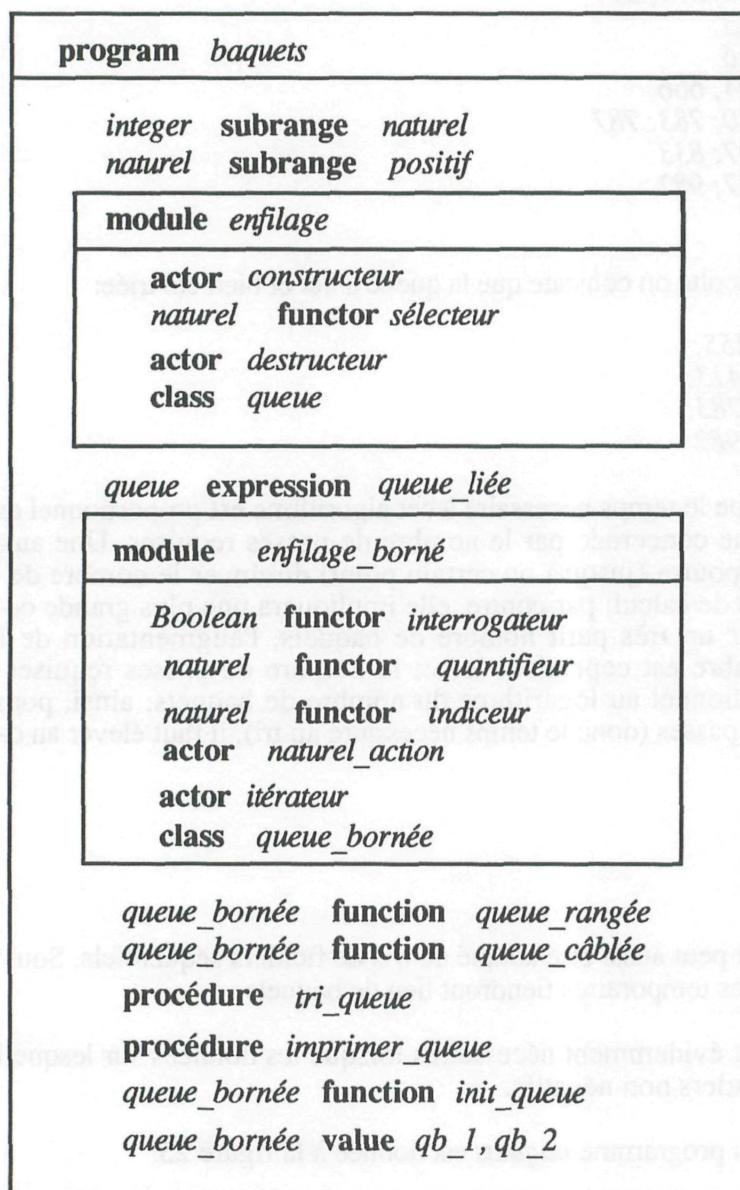

program baquets
integer subrange naturel
naturel subrange positif
module enfilage
actor constructeur
naturel functor sélecteur
actor destructeur
class queue
queue expression queue_liée
module enfilage_borné
Boolean functor interrogateur
naturel functor quantifieur
naturel functor indiceur
actor naturel_action
actor itérateur
class queue_bornée
queue_bornée function queue_rangée
queue_bornée function queue_câblée
procédure tri_queue
procédure imprimer_queue
queue_bornée function init_queue
queue_bornée value qb_1, qb_2

Figure 25

```
baquets                                          Vax Newton Compiler 0.2c4
Page    1
                    Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]BAQUETS.NEW*/ */
     1 PROGRAM baquets DECLARE
     4   integer SUBRANGE naturel(naturel>=0)
    12           SUBRANGE positif(positif>0);
    20
    20 MODULE enfilage
    22   ATTRIBUTE queue,constructeur,selecteur,destructeur
    30 (*Definit la classe protocole  queue  dont les objets sont des queues
    30   de valeurs entieres non negatives.
    30 *)
    30 DECLARE(*enfilage*)
    31   ACTOR constructeur(naturel VALUE val);
    39   naturel FUNCTOR selecteur;
    43   ACTOR destructeur(naturel REFERENCE var);
    51
    51   CLASS queue
    53     VALUE moi
    55     ATTRIBUTE vide,longueur,enfiler,avant,arriere,defiler
    67     (constructeur VALUE cons;
    72      selecteur VALUE avt,arr;
    78      destructeur VALUE des)
    82   (*Un objet du type  queue  est une queue, initialement vide, de valeurs
    82     entieres non negatives. L'implanteur devra fournir, au moyen
    82     d'objets proceduraux, les operations suivantes:
    82
    82       cons[nat]     : appond la valeur  nat
    82       avt EVAL      : resulte en la valeur a l'avant de la queue
    82       arr EVAL      : resulte en la valeur a l'arriere de la queue
    82       des[nat_var]  : enleve de la queue la valeur inseree en premier
    82                       lieu; stocke cette valeur dans la variable
    82                       nat_var
    82   *)
    82   DECLARE(*queue*)
    83     naturel VARIABLE long VALUE longueur:=0;
    91     (*Le nombre de composantes de la queue*)
    91
    91     Boolean EXPRESSION vide=
    95     (*Vrai ssi la queue est vide*)
    95       longueur=0;
    99
    99     queue FUNCTION enfiler
   102       (naturel VALUE val)
   107     (*Insere dans la queue un nouvel element  val ; le resultat est la
   107       queue concernee  moi .
   107     *)
   107     DO(*enfiler*)
   108       long:=SUCC long; cons[val]
   117     TAKE moi DONE(*enfiler*);
   121
   121     naturel EXPRESSION avant=
   125     (*La valeur de l'element insere le plus anciennement dans la queue.
   125
```

```
 125        Condition d'emploi:  ~vide
 125      *)
 125        (IF longueur=0 THEN
 131           print(line,"###avant de queue vide non defini###")
 137         RETURN DONE;
 140        avt EVAL);
 144
 144      naturel EXPRESSION arriere=
 148      (*La valeur de l'element insere le plus recemment dans la queue.
 148
 148        Condition d'emploi: ~vide
 148      *)
 148        (IF longueur=0 THEN
 154           print(line,"###arriere de queue vide non defini###")
 160         RETURN DONE;
 163        arr EVAL);
 167
 167      queue FUNCTION defiler
 170        (naturel REFERENCE var)
 175      (*Elimine de la queue l'element qui y a ete place le plus
 175        anciennement; stocke sa valeur dans la variable reperee par
 175        var ; le resultat est la queue concernee  moi .
 175
 175        Condition d'emploi:  ~vide
 175      *)
 175      DO(*defiler*)
 176        IF longueur=0 THEN
 181          print(line,"###defiler non applicable a queue vide###")
 187        RETURN DONE;
 190        des[var]; long:=PRED long
 199      TAKE moi DONE(*defiler*)
 202    DO(*queue*)DONE
 204 DO(*enfilage*)DONE;
 207
 207    queue EXPRESSION queue_liee=
 211    (*Le resultat est une queue, non bornee a priori, de valeurs du
 211      type  naturel .
 211    *)
 211      DECLARE
 212        OBJECT liste VALUE moi
 216          (naturel VALUE val; liste VARIABLE suiv)
 225        VARIABLE arr:=NIL
 229      DO TAKE
 231        queue(BODY
 234                constructeur(naturel VALUE n)
 240              DECLARE liste VALUE nouv=liste(:n,moi:) DO
 252                CONNECT nouv=:arr THEN
 257                  suiv:=:nouv.suiv
 262                DONE
 263              DONE,
 265              BODY selecteur DO TAKE arr.suiv.val DONE,
 276              BODY selecteur DO TAKE arr.val DONE,
```

```
baquets                                          Vax Newton Compiler 0.2c4
Page    3
                        Source listing

                  BODY
                    destructeur(naturel REFERENCE nn)
                  DO
                    CONNECT arr THEN
                      nn:=suiv.val;
                      UNLESS suiv=moi THEN
                        suiv:=suiv.suiv
                      DEFAULT arr:=NIL DONE
                    DONE
                  DONE)
          DONE(*queue_liee*);
      /* /*EJECT*/ */
```

Comme dans le cas des piles, il est intéressant de distinguer entre les queues dont la longueur maximum est connue à priori et celles de longueur quelconque. Dans le programme *baquets*, la démarche adoptée est semblable à celle du programme *pilistes*. Dans le module *enfilage*, il est implanté la classe protocole *queue* représentant des queues, non bornées à priori, de valeurs entières du type *naturel*. On note à ce sujet que ce programme fait intervenir aussi bien l'identificateur *queue* que le mot-clé **queue**; la chose est possible dès le moment où seule la forme majuscule *QUEUE* est réservée pour la représentation du mot-clé. Ceci implique, par contre, que le programme ne peut être mis automatiquement en forme au moyen du pragmat *NEWSOURCE*; il faut d'emblée le préparer avec la forme réquise majuscule des mots clés et le compiler avec le pragmat *OLDSOURCE*.
Ce module est suivi de la fonction génératrice *queue_liée*. Cette dernière implante un objet du type *queue* au moyen d'une liste circulaire. La variable *arr* y dénote le noeud contenant l'élément à l'arrière de la queue, la variable *arr.suiv* celui à l'avant de la queue; ensuite, les éléments sont chaînés de l'avant vers l'arrière de la queue, c'est-à-dire en sens contraire de ce que suggère la figure 23. Ainsi, une queue dans laquelle on aura inséré successivement les valeurs *19, 87, 33, 79 et 56* aura la représentation donnée à la figure 26.

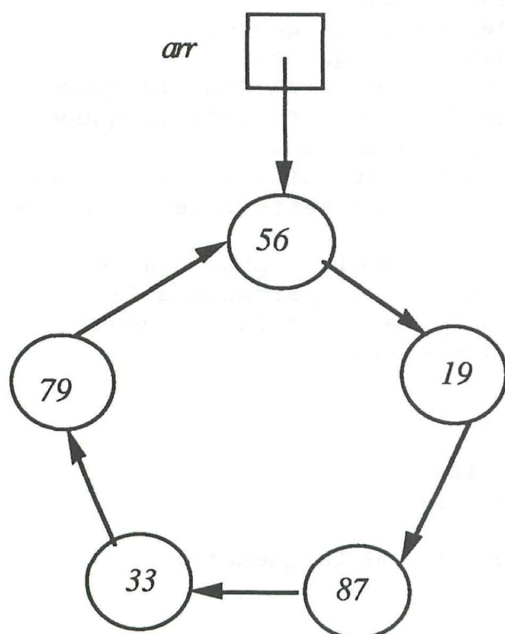


Figure 26

Le lecteur peut facilement se rendre compte que ce sens de parcours est obligatoire; si les liaisons étaient inversées, le destructeur *défiler* ne pourrait être implanté de manière économique.
Dans l'implantation du constructeur *enfiler* , on remarque le générateur connecté *(:n, moi :)*. D'une manière générale un objet défini au moyen d'une déclaration d'objet, de classe ou de processus peut être construit au moyen d'un générateur connecté caractérisé par une liste de paramètres effectifs encadrée des symboles *(:* et *:)*; les attributs de l'objet en cours de création sont disponibles à l'intérieur de la liste des paramètres; dans le cas particulier, c'est le cas de la valeur *moi*: l'élaboration du générateur connecté *(:n, moi :)* aboutit donc à une liste circulaire d'un élément.

```
MODULE enfilage_borne
  ATTRIBUTE queue_bornee,
            interrogateur,quantifieur,indiceur,
            naturel_action,iterateur
DECLARE(*enfilage_borne*)
  Boolean FUNCTOR interrogateur;
  naturel FUNCTOR quantifieur;
  naturel FUNCTOR indiceur(positif VALUE rang);
  ACTOR naturel_action(positif VALUE rang; naturel VALUE n);
  ACTOR iterateur(naturel_action VALUE acte);

  CLASS queue_bornee
    VALUE moi
    INDEX element
    ATTRIBUTE capacite,vide,plein,longueur,
              enfiler,avant,arriere,defiler,parcourir,
              queue_induite
    (positif VALUE capacite;
     interrogateur VALUE vd,pln;
     quantifieur VALUE quant;
     constructeur VALUE cons;
     selecteur VALUE avt,arr;
     indiceur VALUE ind;
     destructeur VALUE des;
     iterateur VALUE parc)
  (*Un objet du type queue_bornee est une queue d'au maximum
    capacite  valeurs entieres non negatives. A tout objet de cette
    classe protocole, l'implanteur devra fournir, au moyen d'objets
    proceduraux, les operations suivantes:

      vd EVAL      : vrai ssi la queue est vide.
      pln EVAL     : vrai ssi la queue est pleine.
      quant EVAL   : le nombre de composantes de la queue.
      cons[nat]    : appond la valeur  nat  a la queue.
      avt EVAL     : la valeur inseree en premier lieu dans la queue.
      arr EVAL     : la valeur inseree en dernier lieu dans la queue.
      ind[rg]      : l'element au rang  rg  de la queue.
      des[nat_var]: enleve de la queue la valeur qui y a ete inseree
                    le plus anciennement; stocke cette valeur dans la
                    variable  nat_var .
      parc[acte]   : effectue l'enonce  acte[rg,elt]  pour chaque
                    element  elt  de la queue; le parcours a lieu
                    depuis l'avant jusqu'a l'arriere de la queue:
                    rg  est le rang de l'element.
  *)
  DECLARE(*queue_bornee*)
    Boolean EXPRESSION
      vide=vd[] (*vrai ssi la queue est vide*),
      plein=pln[] (*vrai ssi la queue est pleine*);
    naturel EXPRESSION
      longueur=quant[] (*le nombre d'elements de la queue*);
```

Source listing

```
  queue VALUE queue_induite=
  (*La queue concernee vue en tant que queue generale*)
    queue(cons, avt, arr, des);

  queue_bornee FUNCTION enfiler
    (naturel VALUE val)
  (*Append a la queue un nouvel element de valeur  val ;
    le resultat est la queue  moi .

    Condition d'emploi:  ~plein
  *)
  DO(*enfiler*)
    IF pln[] THEN
      print(line,"###Enfiler non applicable a queue bornee pleine###")
    RETURN DONE;
    queue_induite.enfiler(val)
  TAKE moi DONE(*enfiler*);

  naturel EXPRESSION
    avant  =queue_induite.avant(*L'element a l'avant de la queue*),
    arriere=queue_induite.arriere(*L'element a l'arriere de la queue*);

  naturel FUNCTION element
    (positif VALUE rg)
  (*L'element de rang  rg  de la queue concernee.

    Condition d'emploi:  rg<=longueur
  *)
  DO(*element*)
    UNLESS rg<=quant[] THEN
      print(line,"###Element de queue inexistant###")
    RETURN DONE
  TAKE ind[rg] DONE(*element*);

  queue_bornee FUNCTION defiler
    (naturel REFERENCE var)
  (*Elimine de la queue concernee l'element qui y a ete insere le
    plus anciennement; le resultat est la queue concernee  moi .

    Condition d'emploi:  ~vide
  *)
  DO queue_induite.defiler(var) TAKE moi DONE(*defiler*);

  queue_bornee FUNCTION parcourir
    (naturel_action VALUE acte)
  (*Parcourt la queue concernee depuis l'avant jusqu'a l'arriere;
    pour chaque element effectue l'enonce  acte[rg,val]  dans
    lequel  rg  est le rang et  val  la valeur de l'element.
    Le resultat est la queue concernee  moi .
  *)
  DO parc[acte] TAKE moi DONE(*parcourir*)
DO(*queue_bornee*)DONE
```

```
DO(*enfilage_borne*)DONE;

  queue_bornee FUNCTION queue_rangee
    (positif VARIABLE capacite VALUE limite)
  (*Realise une implantation d'une  queue_bornee  au moyen d'une
    rangee circulaire et de deux compteurs.
  *)
  DECLARE(*queue_rangee*)
    (*Representation interne*)
    naturel ROW vecteur VALUE tab=vecteur(0 TO capacite);
    integer VARIABLE ins,del:=0
  DO(*queue_rangee*)TAKE
    queue_bornee
      ((positif[SUCC capacite]=:capacite),
       BODY interrogateur DO TAKE ins=del DONE,
       BODY interrogateur DO TAKE (SUCC ins)\limite=del DONE,
       BODY quantifieur DO TAKE (ins-del)\limite DONE,
       BODY
         constructeur(naturel VALUE val)
       DO tab[((SUCC ins)\limite=:ins)]:=val DONE,
       BODY selecteur DO TAKE tab[del] DONE,
       BODY selecteur DO TAKE tab[(PRED ins)\limite] DONE,
       BODY
         indiceur(positif VALUE rg)
       DO TAKE tab[(PRED del+rg)\limite] DONE,
       BODY
         destructeur(naturel REFERENCE var)
       DO var:=tab[((SUCC del)\limite=:del)] DONE,
       BODY
         iterateur(naturel_action VALUE acte)
       DO
         FOR integer VALUE k FROM 1 BY 1 TO (ins-del)\limite REPEAT
           acte[k,tab[(PRED del+k)\limite]]
         REPETITION
       DONE)
  DONE(*queue_rangee*);

  queue_bornee FUNCTION queue_cablee
    (positif VALUE capacite)
  (*Implante un objet du type  queue_bornee  au moyen d'une queue
    predefinie.
  *)
  DECLARE(*queue_cablee*)
    (*Representation interne*)
    naturel QUEUE nat_queue VALUE q=nat_queue(capacite);
    (*Operations predefinies*)
    Boolean EXPRESSION vide=EMPTY q,
                       plein=FULL q;
    naturel EXPRESSION longueur=CARD q;

    PROCEDURE appondre(naturel VALUE n) DO
```

```
     q APPEND n
  DONE(*appondre*);

  naturel EXPRESSION avant  =q FRONT,
                    arriere=q BACK;

  naturel FUNCTION element
     (positif VALUE rg)
  DO TAKE q[rg] DONE(*element*);

  PROCEDURE dependre(naturel REFERENCE nn) DO
     nn FROM q
  DONE(*dependre*);

  PROCEDURE parcourir(naturel_action VALUE acte) DO
     THROUGH q
        INDEX rg
        VALUE q_rg
     REPEAT acte[rg,q_rg] REPETITION
  DONE(*parcourir*)
DO(*queue_cablee*)TAKE
  queue_bornee
     (CAPACITY q,
      interrogateur(vide),interrogateur(plein),
      quantifieur(longueur),
      constructeur(appondre),
      selecteur(avant),selecteur(arriere),
      indiceur(element),
      destructeur(dependre),
      iterateur(parcourir))
DONE(*queue_cablee*);
/* /*EJECT*/ */
```

```
baquets                                        Vax Newton Compiler 0.2c4
Page     8
                  Source listing

  982    positif SUBRANGE base_numeration(base_numeration>1);
  991
  991    PROCEDURE tri_queue
  993      (queue VALUE q; base_numeration VALUE base)
 1002    (*Trie, par ordre croissant, la queue  q ; utilise un algorithme
 1002      de tri incluant  base  baquets.
 1002    *)
 1002    DECLARE(*tri_queue*)
 1003      queue ROW queue_rangee VALUE baquets=
 1009        THROUGH queue_rangee(0 TO PRED base):=queue_liee REPETITION;
 1021      naturel VARIABLE limite:=0,n; positif VARIABLE facteur:=1
 1034    DO(*tri_queue*)
 1035      (*Fait une premiere semaille*)
 1035      UNTIL q.vide REPEAT
 1040        q.defiler(n); limite:=limite MAX n;
 1053        baquets[n\base].enfiler(n)
 1064      REPETITION;
 1066      UNTIL
 1067        (*Reconstruit la queue*)
 1067        THROUGH baquets VALUE bac REPEAT
 1072          UNTIL bac.vide REPEAT
 1077            bac.defiler(n); q.enfiler(n)
 1090          REPETITION
 1091        REPETITION
 1092      TAKE facteur>limite%base REPEAT
 1099        facteur:=facteur*base;
 1105        (*Effectue une nouvelle semaille*)
 1105        UNTIL q.vide REPEAT
 1110          q.defiler(n);
 1117          baquets[n%facteur\base].enfiler(n)
 1130        REPETITION
 1131      REPETITION
 1132    DONE(*tri_queue*);
 1134
 1134    PROCEDURE imprimer_queue
 1136      (queue_bornee VALUE qb)
 1141    DO(*imprimer_queue*)
 1142      print(line,"***Impression d'une queue***",
 1148            line,___"capacite:"_,qb.capacite,
 1158            line,___"longueur:"_,qb.longueur,
 1168            line,___"queue vide?"_,qb.vide,
 1178            line,___"queue pleine?"_,qb.plein,
 1188            line,"***Liste des elements***");
 1193      qb.parcourir
 1196        (BODY
 1198           naturel_action(positif VALUE rang; naturel VALUE elt)
 1208         DO
 1209           IF rang\5=1 THEN line DONE; edit(elt,12,0)
 1227         DONE);
 1230       print(line,"<<<FIN>>>")
 1236    DONE(*imprimer_queue*);
 1238
```

```
baquets                                         Vax Newton Compiler 0.2c4
Page     9
                     Source listing

     naturel FUNCTOR suite(positif VALUE rang);

     queue_bornee FUNCTION init_queue
       (queue_bornee VALUE qb; suite VALUE s)
     (*Initialise la queue bornee  qb ; l'element de rang  k  sera
       egal a  s[k]

       Condition d'emploi: qb.vide
     *)
     DO(*init_queue*)
       UNLESS qb.vide THEN
         print(line,"###queue deja initialisee###")
       RETURN DONE;
       FOR integer VALUE k FROM 1 TO qb.capacite REPEAT
         qb.enfiler(s[k])
       REPETITION
     TAKE qb DONE(*init_queue*);
     /* /*EJECT*/ */
```

Le module *enfilage_borné* inclut la classe protocole *queue_bornée* décrivant l'interface de queues de valeurs entières non négatives d'une capacité bornée à priori; de nouveau, au moyen de l'attribut *queue_induite*, il est possible d'accéder au même objet vu comme une instance du type *queue*. Une queue bornée hérite de la queue induite qui lui est associée les attributs *enfiler*, *avant*, *arrière* et *défiler*.

Il a été construit les deux fonctions génératrices *queues_rangée* et *queue_câblée*. La seconde implante directement une queue bornée au moyen d'un objet *q* du type *naturel* **queue** *nat_queue*; le première réalise un modèle d'implantation d'un type queue.

La représentation interne choisie est une rangée *tab* de *capacité + 1* variables du type de base *naturel* et de deux compteurs auxiliaires *ins* et *del*. La rangée *tab* est gérée circulairement: la variable *ins* a pour valeur le nombre d'éléments insérés dans la queue modulo *limite* = *capacité* + *1* et la variable *del*, le nombre d'éléments éliminés de la queue modulo *limite*.

Le lecteur peut constater que si l'on avait omis l'élément supplémentaire dans la rangée représentative *tab*, l'implantation des interrogateurs *vide* et *plein* aurait donné lieu à la même expression.

```
baquets                                    Vax Newton Compiler 0.2c4
Page   10
                    Source listing

       queue_bornee VALUE
        qb_1=init_queue
                 (queue_rangee(200),
                  BODY suite DO TAKE FLOOR(integer MAX*random) DONE),
        qb_2=init_queue
                 (queue_cablee(250),
                  BODY
                    suite(positif VALUE n)
                  DO TAKE (n+123)**3\65537 DONE)
     DO(*baquets*)
       print("*****Premiere queue*****");
       imprimer_queue(qb_1);
       print(page,"*****Queue triee*****");
       tri_queue(qb_1.queue_induite,10);
       imprimer_queue(qb_1);
       print(page,"*****Deuxieme queue*****");
       imprimer_queue(qb_2);
       print(page,"*****Queue triee*****");
       tri_queue(qb_2.queue_induite,100);
       imprimer_queue(qb_2)
     DONE(*baquets*)

       ****  No messages were issued  ****
```

L'algorithme de tri par baquets est inclus dans la procédure *tri_queue*. Il lui est fourni en paramètre la queue à trier et le nombre de baquets à utiliser. Lors de la première semaille, la valeur du plus grand élément de la queue concernée *q* est obtenue dans la variable *limite*. Cette valeur permettra de savoir à quel moment le tri est achevé. La variable *facteur* est utilisée pour extraire le chiffre approprié des éléments à trier lors des opérations de semaille: cette variable a pour valeur *base* ** *k* où *k* est le nombre de passes déjà (complètement) réalisées. Il s'ensuit que le chiffre approprié d'un élément de valeur *n* peut être obtenu au moyen de l'expression *n % facteur* \ *base*. Dès que *facteur > limite* , *le quotient n % facteur* sera nécessairement nul; les cas échéant, tous les éléments seraient placés dans le baquet *0* puis réinsérés dans le même ordre dans *q*: ceci implique qu'en fait le tri est terminé. Il est préférable de ne pas tester directement la relation *facteur > limite*; en effet, si la valeur *limite* est grande, il se pourrait que la plus petite valeur de *facteur* qui dépasse *limite* soit supérieure à *integer* **max** et l'algorithme serait en défaut. Il est facile de remédier à cet inconvénient. A la fin d'une passe de tri, et avant du multiplier la variable *facteur* par *base* pour la passe suivante, il est examiné si le produit *facteur* * *base* dépassera *limite*. Pour la raison évoquée plus haut, ce test est fait au moyen de l'expression *facteur > limite % base* et non *facteur * base > limite*.

Le premier paramètre de la procédure *tri_queue* doit être une queue générale du type *queue*; pour trier les queues bornées *qb_1* et *qb_2*, il faut donc lui communiquer les objets *qb_1.queue_induite* et *qb_2.queue_induite*. La queue *qb_1* a été triée au moyen de *10* baquets; certains éléments de *qb_1* ont dix chiffres décimaux: ceci implique que dix passes auront été nécessaires pour ce tri. Avec *100* baquets, il aura suffi de trois passes pour trier *qb_2* puisque son plus grand élément *65505* a manifestement trois chiffres dans la base cent. En fait, pour ce dernier tri, la même efficacité aurait été atteinte avec seulement *41* baquets puisque *41 ** 3 = 68921 > 65505* (par contre, avec *40* baquets, une passe supplémentaire serait nécessaire puisque *40 ** 3 = 64000 < 65505*). On vérifie qu'au moins *256* baquets seraient nécessaires pour réduire de trois à deux le nombre de passes nécessaires au tri de la queue *qb_2*; ainsi avec un nombre de baquets quelconque entre *41* et *255*, l'efficacité de ce tri sera la même: ceci illustre bien la lenteur de l'augmentation de l'efficacité de l'algorithme en fonction du nombre de baquets lorsque ce nombre devient (relativement) grand.

```
*****Premiere queue*****
***Impression d'une queue***
   capacite:            200
   longueur:            200
   queue vide?  'FALSE'
   queue pleine?   'TRUE'
***Liste des elements***
  1465041833  1923021397  1414189379  1233497179  1247438787
   367900536   619456196   819445295  1695337783   406542307
  1515097255   966581982   608712010  1208451666  2134787780
   486781604   108146807  1949998947  1502299413  1962309249
   599689296   387524938   986785226  1790692237   499858720
  2050094786  1371747616   251123127   698134283  1862010745
  1059909067  1369219803   188567865   494976560  1484575748
   499010183  1850791654   370936776    68118270   820658034
  1009435744  2030466585  1289747933   958429424  1699721054
   225655044  1633125637   633331274   652314861  1684699903
  1326881141  1781712767   231864311  1956838025  1594957568
   606897093   734713861  2123756503  1175370774   391661554
   783723278   954449342   891296366  1514187378   766038252
  1332795423  1190053816   602905990  1843097095  1605366134
   357076760  2031648940  2126098651   424257432  1400753194
   209929926  1023748804  1855285193  1594187835  1381213292
   330112548  1702001380   628582878  1084972779   357153786
   317488964  1928503157   780380651  1652312301   928554314
   910011511  1944205808   600646310   558885190  1199023557
   727751824  2130834557   780319614  1254714276  1580572191
  1822545176   487250820   837205453   379829518  1089256310
   165245691   111532597   694716560   904509761   137105428
   537401911  1861319330   698468162   646526892  1218462848
   573339986   365863461  2103584727   805718809  1410585167
   861433900  1815982320  1331025636   985725249   152251443
   171663922   265134395  1857059117     4746925  1135069735
  1637169154  1070055366   999304195   195810325  1625470013
   716691755  1903234543   808251858  1784519173  1639521068
  1593827956   600869733   457990890  1010681797  1476234416
  1376742368  1990771847   428923244    80435709   827854978
   759436080  1219924643  1954517120    73959453  1541577951
  1234420830  1966742113  1026633366  1886717725  1476474871
  1289254170  1273474131  1936411743   721395691  1889748789
  1389768567   682462923   228084934   214818122  1062368234
  1156751902   857100302   428927171  1307186848  1759299324
  1629692880  1639970696  1573895550   547730569  1535856431
   506622545  1697553427  2113284984   223776457  2104131132
   167154239  1437963952  2092725269  1442202389  1552105324
   313318641   237038988  1151842095  1523520476  1659710346
  1591165514   822590408  1513373422  1118739574    87066487
<<<FIN>>>
```

```
*****Queue triee*****
***Impression d'une queue***
   capacite:           200
   longueur:           200
   queue vide?  'FALSE'
   queue pleine?   'TRUE'
***Liste des elements***
     4746925    68118270    73959453    80435709    87066487
   108146807   111532597   137105428   152251443   165245691
   167154239   171663922   188567865   195810325   209929926
   214818122   223776457   225655044   228084934   231864311
   237038988   251123127   265134395   313318641   317488964
   330112548   357076760   357153786   365863461   367900536
   370936776   379829518   387524938   391661554   406542307
   424257432   428923244   428927171   457990890   486781604
   487250820   494976560   499010183   499858720   506622545
   537401911   547730569   558885190   573339986   599689296
   600646310   600869733   602905990   606897093   608712010
   619456196   628582878   633331274   646526892   652314861
   682462923   694716560   698134283   698468162   716691755
   721395691   727751824   734713861   759436080   766038252
   780319614   780380651   783723278   805718809   808251858
   819445295   820658034   822590408   827854978   837205453
   857100302   861433900   891296366   904509761   910011511
   928554314   954449342   958429424   966581982   985725249
   986785226   999304195  1009435744  1010681797  1023748804
  1026633366  1059909067  1062368234  1070055366  1084972779
  1089256310  1118739574  1135069735  1151842095  1156751902
  1175370774  1190053816  1199023557  1208451666  1218462848
  1219924643  1233497179  1234420830  1247438787  1254714276
  1273474131  1289254170  1289747933  1307186848  1326881141
  1331025636  1332795423  1369219803  1371747616  1376742368
  1381213292  1389768567  1400753194  1410585167  1414189379
  1437963952  1442202389  1465041833  1476234416  1476474871
  1484575748  1502299413  1513373422  1514187378  1515097255
  1523520476  1535856431  1541577951  1552105324  1573895550
  1580572191  1591165514  1593827956  1594187835  1594957568
  1605366134  1625470013  1629692880  1633125637  1637169154
  1639521068  1639970696  1652312301  1659710346  1684699903
  1695337783  1697553427  1699721054  1702001380  1759299324
  1781712767  1784519173  1790692237  1815982320  1822545176
  1843097095  1850791654  1855285193  1857059117  1861319330
  1862010745  1886717725  1889748789  1903234543  1923021397
  1928503157  1936411743  1944205808  1949998947  1954517120
  1956838025  1962309249  1966742113  1990771847  2030466585
  2031648940  2050094786  2092725269  2103584727  2104131132
  2113284984  2123756503  2126098651  2130834557  2134787780
<<<FIN>>>
```

```
*****Deuxieme queue*****
***Impression d'une queue***
   capacite:           250
   longueur:           250
   queue vide?  'FALSE'
   queue pleine?   'TRUE'
***Liste des elements***
        6051        52552        34266        16736        65505
       49505        34279        19833         6173        58842
       46772        35506        25050        15410         6592
       64139        56983        50667        45197        40579
       36819        33923        31897        30747        30479
       31099        32613        35027        38347        42579
       47729        53803        60807         3210        12092
       21922        32706        44450        57160         5305
       19965        35609        52243         4336        22968
       42608        63262        19399        42099          294
       25064        50878        12205        40125         3570
       33620        64744        31411        64701        33546
        3489        40073        12230        51040        25435
         958        43152        20949        65429        45524
       26777         9194        58318        43081        29026
       16159         4486        59550        50283        42228
       35391        29778        25395        22248        20343
       19686        20283        22140        25263        29658
       35331        42288        50535        60078         5386
       17539        31006        45793        61906        13814
       32597        52724         8664        31497        55692
       15718        42655         5435        35138          696
       33189         1549        36856         8042        46187
       20223        61230        38140        16496        61841
       43107        25837        10037        61250        48408
       37054        27194        18834        11980         6638
        2814          514        65281          510         2818
        6674        12084        19054        27590        37698
       49384        62654        11977        28433        46491
         620        21900        44800         3789        29947
       57743        21646        52736        19945        54353
       24892        62642        36535        12114        54922
       33891        14564        62484        46583        32404
       19953         9236          259        58565        53086
       49365        47408        47221        48810        52181
       57340        64293         7509        18068        30439
       44628        60641        12947        32626        54147
       11979        37202        64285        27697        58518
       25680        60263        31199         4031        44302
       20944        65037        45513        27915        12249
       64058        52274        42440        34562        28646
       24698        22724        22730        24722        28706
       34688        42674        52670        64682        13179
       29241        47337         1936        24118        48352
        9107        37463         2352        34854         3901
       40573        13802        54668        32103        11650
       58852        42641        28560        16615         6812
       64694        59193        55852        54677        55674
       58849        64208         6220        15965        27912
       42067        58436        11488        32303        55350
<<<FIN>>>
```

```
*****Queue triee*****
***Impression d'une queue***
   capacite:          250
   longueur:          250
   queue vide?  'FALSE'
   queue pleine?   'TRUE'
***Liste des elements***
         259        294        510        514        620
         696        958       1549       1936       2352
        2814       2818       3210       3489       3570
        3789       3901       4031       4336       4486
        5305       5386       5435       6051       6173
        6220       6592       6638       6674       6812
        7509       8042       8664       9107       9194
        9236      10037      11488      11650      11977
       11979      11980      12084      12092      12114
       12205      12230      12249      12947      13179
       13802      13814      14564      15410      15718
       15965      16159      16496      16615      16736
       17539      18068      18834      19054      19399
       19686      19833      19945      19953      19965
       20223      20283      20343      20944      20949
       21646      21900      21922      22140      22248
       22724      22730      22968      24118      24698
       24722      24892      25050      25064      25263
       25395      25435      25680      25837      26777
       27194      27590      27697      27912      27915
       28433      28560      28646      28706      29026
       29241      29658      29778      29947      30439
       30479      30747      31006      31099      31199
       31411      31497      31897      32103      32303
       32404      32597      32613      32626      32706
       33189      33546      33620      33891      33923
       34266      34279      34562      34688      34854
       35027      35138      35331      35391      35506
       35609      36535      36819      36856      37054
       37202      37463      37698      38140      38347
       40073      40125      40573      40579      42067
       42099      42228      42288      42440      42579
       42608      42641      42655      42674      43081
       43107      43152      44302      44450      44628
       44800      45197      45513      45524      45793
       46187      46491      46583      46772      47221
       47337      47408      47729      48352      48408
       48810      49365      49384      49505      50283
       50535      50667      50878      51040      52181
       52243      52274      52552      52670      52724
       52736      53086      53803      54147      54353
       54668      54677      54922      55350      55674
       55692      55852      56983      57160      57340
       57743      58318      58436      58518      58565
       58842      58849      58852      59193      59550
       60078      60263      60641      60807      61230
       61250      61841      61906      62484      62642
       62654      63262      64058      64139      64208
       64285      64293      64682      64694      64701
       64744      65037      65281      65429      65505
<<<FIN>>>
```

Des listes de nature très générale peuvent être implantées au moyen de listes circulaires bidirectionnelles. A titre d'exemple, on considère les déclarations suivantes:

```
objet liste
   (real value val;
      liste variable prec, suiv);

procedure nouer
   (liste  value p, q)
do (* nouer *)
   (p.suiv :=: q.prec.suiv) .prec :=: q.prec
done (*nouer *)
```

La procédure *nouer* permet de faciliter aussi bien l'insertion d'éléments nouveaux dans une liste circulaire bidirectionnelle que l'élimination d'éléments d'une telle liste. En-effet, on vérifie qu'appliquée à deux noeuds *p* et *q* appartenant à deux listes circulaires distinctes, elle fusionne ces deux listes en une; inversement, appliquée à deux noeuds *p* et *q* d'une même liste circulaire, elle scindera cette liste en deux sauf si $q = p.suiv$ (dans ce dernier cas son effet est vide). Le premier cas est illustré dans les figures 27 et 28; dans ces figures (ainsi que dans les deux suivantes), on suppose que les liaisons *suiv* sont orientées dans le sens des aiguilles de la montre et les liaisons *prec* dans le sens contraire à ce dernier.

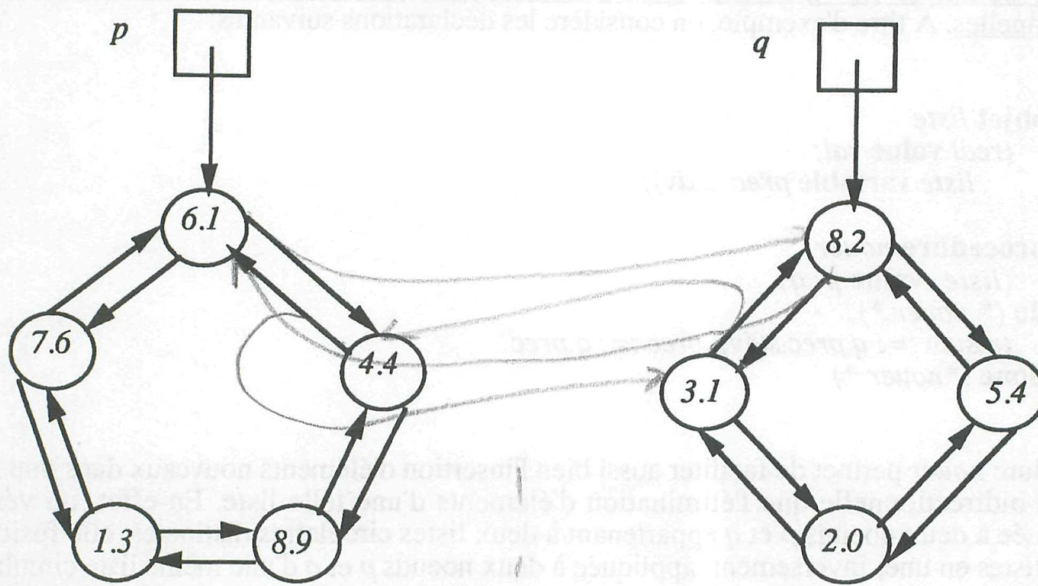


Figure 27

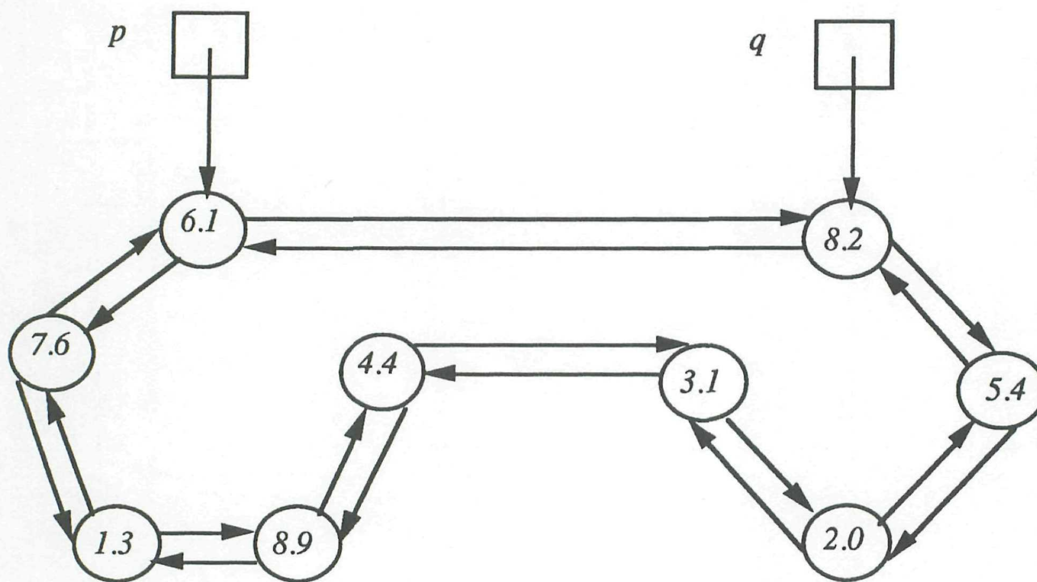
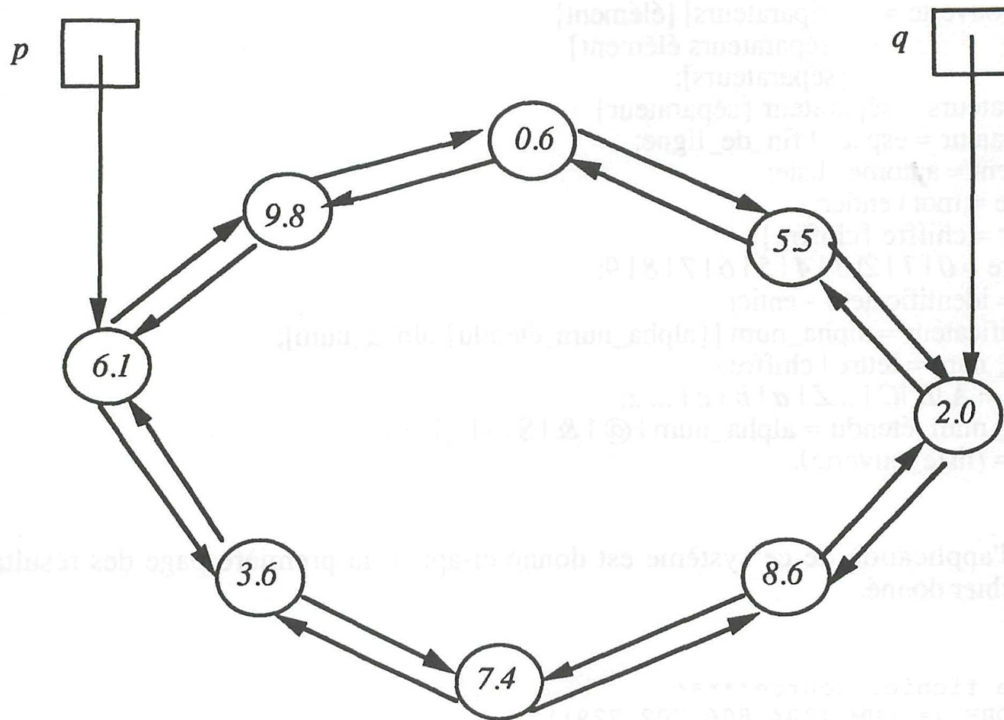


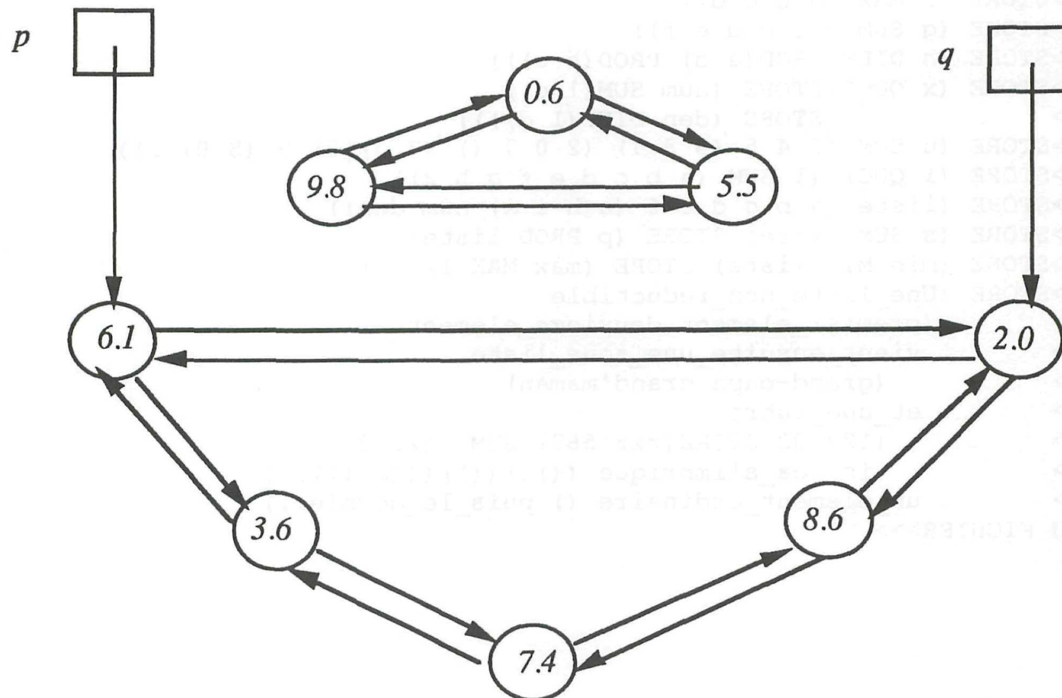
Figure 28

Le deuxième cas est illustré dans les figures 29 et 30. En résumé, dans le premier cas, la liste *q* est incorporée à la liste *p* entre les éléments *p* et *p.suiv*, ou (ce qui revient au même) la liste *p* est incorporée à la liste *q* entre les éléments *q.prec* et *q*.



(avant *nouer (p, q)*)

Figure 29



(après *nouer (p, q)*)

Figure 30

Dans le deuxième cas, toute la sous-liste située entre les éléments *p* et *q* est éliminée et constituée en une liste circulaire séparée. On va illustrer cette technique dans un interprète de traitement de liste minimal. Cet interprète accepte, comme donnée, une liste d'entités spécifiées au moyen des règles de production suivantes:

liste_ouverte = [séparateurs] [élément]
{séparateurs élément}
[séparateurs];
séparateurs = séparateur {séparateur}
séparateur = espace | fin_de_ligne;
élément = atome | liste;
atome = mot | entier;
entier = chiffre {chiffre};
chiffre = *0* | *1* | *2*| *3* | *4* | *5* | *6* | *7* | *8* | *9*;
mot = identificateur - entier
identificateur = alpha_num [{alpha_num_étendu} alpha_num];
alpha_num = lettre | chiffre;
lettre = *A* |*B* |*C* | ...*Z* | *a* | *b* | *c* | ... *z*;
alpha_num_étendu = alpha_num | @ | & | $ | - | _ | . |' ;
liste = *(*liste_ouverte*)*.

Un exemple d'application de ce système est donné ci-après; la première page des résultats reproduit le fichier donné.

```
*****Listage fichier source*****
    1--->STORE (a SUM (204 806 202 729))
    2--->STORE (b PROD (802 a 35))
    3--->STORE (c DIFF (a b))
    4--->STORE (d QUOT (a b))
    5--->STORE (e MIN (a b c d))
    6--->STORE (f MAX (a b c d))
    7--->STORE (g SUM(a b c d e f))
    8--->STORE (h DIFF(PROD(a d) PROD(b c)))
    9--->STORE (x QUOT(STORE (num SUM(1 d))
   10--->              STORE (den DIFF(1 d))))
   11--->STORE (u SUM (3 4 5 (9 8 7) (2 0 7 () (7 3) 8) 9 (8 9) 5))
   12--->STORE (i QUOT (1 SUM (a b c d e f g h x)))
   13--->STORE (liste (a b c d e f (u h i x) num den))
   14--->STORE (s SUM liste) STORE (p PROD liste)
   15--->STORE (min MIN liste) STORE (max MAX liste)
   16--->STORE (Une_liste_non_reductible
   17--->        (premier_element deuxieme_element
   18--->         vient_ensuite_une_sous_liste
   19--->           (grand-papa grand'maman)
   20--->         et_une_autre
   21--->           (127 32 STORE(zzz 567) SUM (zzz 2)
   22--->            ici_ca_s'imbrique (()()(())(((a 1)))))
   23--->         un_element_ordinaire () puis_le_dernier))
<<<FIN DU FICHIER>>>
```

Le système donne ensuite un listage structuré de la liste obtenue.

```
***Liste obtenue***
***Impression d'une liste***
  1: mot(STORE)
  2: sous-liste
  ***Impression d'une liste***
    1: mot(a)
    2: mot(SUM)
    3: sous-liste
    ***Impression d'une liste***
      1: entier(204)
      2: entier(806)
      3: entier(202)
      4: entier(729)
    <<<FIN>>>
  <<<FIN>>>
  3: mot(STORE)
  4: sous-liste
  ***Impression d'une liste***
    1: mot(b)
    2: mot(PROD)
    3: sous-liste
    ***Impression d'une liste***
      1: entier(802)
      2: mot(a)
      3: entier(35)
    <<<FIN>>>
  <<<FIN>>>
  5: mot(STORE)
  6: sous-liste
  ***Impression d'une liste***
    1: mot(c)
    2: mot(DIFF)
    3: sous-liste
    ***Impression d'une liste***
      1: mot(a)
      2: mot(b)
    <<<FIN>>>
  <<<FIN>>>
  7: mot(STORE)
  8: sous-liste
  ***Impression d'une liste***
    1: mot(d)
    2: mot(QUOT)
    3: sous-liste
    ***Impression d'une liste***
      1: mot(a)
      2: mot(b)
    <<<FIN>>>
  <<<FIN>>>
  9: mot(STORE)
 10: sous-liste
  ***Impression d'une liste***
    1: mot(e)
    2: mot(MIN)
    3: sous-liste
    ***Impression d'une liste***
      1: mot(a)
      2: mot(b)
      3: mot(c)
```

```
        4: mot(d)
      <<<FIN>>>
    <<<FIN>>>
  11: mot(STORE)
  12: sous-liste
    ***Impression d'une liste***
      1: mot(f)
      2: mot(MAX)
      3: sous-liste
      ***Impression d'une liste***
        1: mot(a)
        2: mot(b)
        3: mot(c)
        4: mot(d)
      <<<FIN>>>
    <<<FIN>>>
  13: mot(STORE)
  14: sous-liste
    ***Impression d'une liste***
      1: mot(g)
      2: mot(SUM)
      3: sous-liste
      ***Impression d'une liste***
        1: mot(a)
        2: mot(b)
        3: mot(c)
        4: mot(d)
        5: mot(e)
        6: mot(f)
      <<<FIN>>>
    <<<FIN>>>
  15: mot(STORE)
  16: sous-liste
    ***Impression d'une liste***
      1: mot(h)
      2: mot(DIFF)
      3: sous-liste
      ***Impression d'une liste***
        1: mot(PROD)
        2: sous-liste
        ***Impression d'une liste***
          1: mot(a)
          2: mot(d)
        <<<FIN>>>
        3: mot(PROD)
        4: sous-liste
        ***Impression d'une liste***
          1: mot(b)
          2: mot(c)
        <<<FIN>>>
      <<<FIN>>>
    <<<FIN>>>
  17: mot(STORE)
  18: sous-liste
    ***Impression d'une liste***
      1: mot(x)
      2: mot(QUOT)
      3: sous-liste
      ***Impression d'une liste***
        1: mot(STORE)
        2: sous-liste
        ***Impression d'une liste***
          1: mot(num)
          2: mot(SUM)
```

```
        3: sous-liste
        ***Impression d'une liste***
          1: entier(1)
          2: mot(d)
        <<<FIN>>>
      <<<FIN>>>
      3: mot(STORE)
      4: sous-liste
      ***Impression d'une liste***
        1: mot(den)
        2: mot(DIFF)
        3: sous-liste
        ***Impression d'une liste***
          1: entier(1)
          2: mot(d)
        <<<FIN>>>
      <<<FIN>>>
    <<<FIN>>>
  <<<FIN>>>
19: mot(STORE)
20: sous-liste
  ***Impression d'une liste***
    1: mot(u)
    2: mot(SUM)
    3: sous-liste
    ***Impression d'une liste***
      1: entier(3)
      2: entier(4)
      3: entier(5)
      4: sous-liste
      ***Impression d'une liste***
        1: entier(9)
        2: entier(8)
        3: entier(7)
      <<<FIN>>>
      5: sous-liste
      ***Impression d'une liste***
        1: entier(2)
        2: entier(0)
        3: entier(7)
        4: sous-liste
        ***Impression d'une liste***
        <<<FIN>>>
        5: sous-liste
        ***Impression d'une liste***
          1: entier(7)
          2: entier(3)
        <<<FIN>>>
        6: entier(8)
      <<<FIN>>>
      6: entier(9)
      7: sous-liste
      ***Impression d'une liste***
        1: entier(8)
        2: entier(9)
      <<<FIN>>>
      8: entier(5)
    <<<FIN>>>
  <<<FIN>>>
21: mot(STORE)
22: sous-liste
  ***Impression d'une liste***
    1: mot(i)
    2: mot(QUOT)
```

```
    3: sous-liste
    ***Impression d'une liste***
      1: entier(1)
      2: mot(SUM)
      3: sous-liste
      ***Impression d'une liste***
        1: mot(a)
        2: mot(b)
        3: mot(c)
        4: mot(d)
        5: mot(e)
        6: mot(f)
        7: mot(g)
        8: mot(h)
        9: mot(x)
      <<<FIN>>>
    <<<FIN>>>
  <<<FIN>>>
23: mot(STORE)
24: sous-liste
  ***Impression d'une liste***
    1: mot(liste)
    2: sous-liste
    ***Impression d'une liste***
      1: mot(a)
      2: mot(b)
      3: mot(c)
      4: mot(d)
      5: mot(e)
      6: mot(f)
      7: sous-liste
      ***Impression d'une liste***
        1: mot(u)
        2: mot(h)
        3: mot(i)
        4: mot(x)
      <<<FIN>>>
      8: mot(num)
      9: mot(den)
    <<<FIN>>>
  <<<FIN>>>
25: mot(STORE)
26: sous-liste
  ***Impression d'une liste***
    1: mot(s)
    2: mot(SUM)
    3: mot(liste)
  <<<FIN>>>
27: mot(STORE)
28: sous-liste
  ***Impression d'une liste***
    1: mot(p)
    2: mot(PROD)
    3: mot(liste)
  <<<FIN>>>
29: mot(STORE)
30: sous-liste
  ***Impression d'une liste***
    1: mot(min)
    2: mot(MIN)
    3: mot(liste)
  <<<FIN>>>
31: mot(STORE)
32: sous-liste
```

```
 ***Impression d'une liste***
   1: mot(max)
   2: mot(MAX)
   3: mot(liste)
 <<<FIN>>>
33: mot(STORE)
34: sous-liste
 ***Impression d'une liste***
   1: mot(Une_liste_non_reductible)
   2: sous-liste
   ***Impression d'une liste***
     1: mot(premier_element)
     2: mot(deuxieme_element)
     3: mot(vient_ensuite_une_sous_liste)
     4: sous-liste
     ***Impression d'une liste***
       1: mot(grand-papa)
       2: mot(grand'maman)
     <<<FIN>>>
     5: mot(et_une_autre)
     6: sous-liste
     ***Impression d'une liste***
       1: entier(127)
       2: entier(32)
       3: mot(STORE)
       4: sous-liste
       ***Impression d'une liste***
         1: mot(zzz)
         2: entier(567)
       <<<FIN>>>
       5: mot(SUM)
       6: sous-liste
       ***Impression d'une liste***
         1: mot(zzz)
         2: entier(2)
       <<<FIN>>>
       7: mot(ici_ca_s'imbrique)
       8: sous-liste
       ***Impression d'une liste***
         1: sous-liste
         ***Impression d'une liste***
         <<<FIN>>>
         2: sous-liste
         ***Impression d'une liste***
         <<<FIN>>>
         3: sous-liste
         ***Impression d'une liste***
           1: sous-liste
           ***Impression d'une liste***
           <<<FIN>>>
         <<<FIN>>>
         4: sous-liste
         ***Impression d'une liste***
           1: sous-liste
           ***Impression d'une liste***
             1: sous-liste
             ***Impression d'une liste***
               1: mot(a)
               2: entier(1)
             <<<FIN>>>
           <<<FIN>>>
         <<<FIN>>>
       <<<FIN>>>
     <<<FIN>>>
```

```
    7: mot(un_element_ordinaire)
    8: sous-liste
    ***Impression d'une liste***
    <<<FIN>>>
    9: mot(puis_le_dernier)
  <<<FIN>>>
 <<<FIN>>>
<<<FIN>>>
<<<FICHIER DONNE A ETE ENTIEREMENT TRAITE>>>
```

La liste fait ensuite l'objet d'une réduction: certains identificateurs dénotent des opérations. Les opérations suivantes sont prévues:

STORE	(variable élément*)*
SUM	désignation_de_liste
PROD	désignation_de_liste
MIN	désignation_de_liste
MAX	désignation_de_liste
DIFF	désignation_de_liste_a_deux_éléments
QUOT	désignation_de_liste_a_deux_éléments

avec:

variable = identificateur;
désignation_de_liste_a_deux_éléments = *(*élément élément*)* | variable.

Dans le cas présent, la réduction a donnée lieu à la liste suivante.

```
***Liste transformee***
***Impression d'une liste***
  1: nombre(  1941.0000000000)
  2: nombre(   54483870.00000)
  3: nombre(  -54481929.00000)
  4: nombre( +.35625222657641610&-04)
  5: nombre(  -54481929.00000)
  6: nombre(   54483870.00000)
  7: nombre(   5823.000035625)
  8: nombre( +.29683863369852300&+16)
  9: nombre(  1.0000712529837)
 10: nombre(   94.00000000000)
 11: nombre( +.33688337246953581&-15)
 12: sous-liste
  ***Impression d'une liste***
    1: nombre(  1941.0000000000)
    2: nombre(   54483870.00000)
    3: nombre(  -54481929.00000)
    4: nombre( +.35625222657641610&-04)
    5: nombre(  -54481929.00000)
    6: nombre(   54483870.00000)
    7: sous-liste
    ***Impression d'une liste***
      1: nombre(   94.00000000000)
      2: nombre( +.29683863369852300&+16)
      3: nombre( +.33688337246953581&-15)
      4: nombre(  1.0000712529837)
    <<<FIN>>>
    8: nombre(  1.0000356252227)
    9: nombre(   .9999643747773)
  <<<FIN>>>
 13: nombre( +.29683863369911500&+16)
 14: nombre( +.57277330308252822&+32)
```

```
15: nombre( -54481929.00000)
16: nombre( +.29683863369852300&+16)
17: sous-liste
 ***Impression d'une liste***
   1: mot(premier_element)
   2: mot(deuxieme_element)
   3: mot(vient_ensuite_une_sous_liste)
   4: sous-liste
   ***Impression d'une liste***
     1: mot(grand-papa)
     2: mot(grand'maman)
   <<<FIN>>>
   5: mot(et_une_autre)
   6: sous-liste
   ***Impression d'une liste***
     1: entier(127)
     2: entier(32)
     3: entier(567)
     4: nombre(   569.0000000000)
     5: mot(ici_ca_s'imbrique)
     6: sous-liste
     ***Impression d'une liste***
       1: sous-liste
       ***Impression d'une liste***
       <<<FIN>>>
       2: sous-liste
       ***Impression d'une liste***
       <<<FIN>>>
       3: sous-liste
       ***Impression d'une liste***
         1: sous-liste
         ***Impression d'une liste***
         <<<FIN>>>
       <<<FIN>>>
       4: sous-liste
       ***Impression d'une liste***
         1: sous-liste
         ***Impression d'une liste***
           1: sous-liste
           ***Impression d'une liste***
             1: nombre(  1941.0000000000)
             2: entier(1)
           <<<FIN>>>
         <<<FIN>>>
       <<<FIN>>>
     <<<FIN>>>
   <<<FIN>>>
   7: mot(un_element_ordinaire)
   8: sous-liste
   ***Impression d'une liste***
   <<<FIN>>>
   9: mot(puis_le_dernier)
 <<<FIN>>>
<<<FIN>>>
```

On peut donc y examiner la nature des réduction faites.

Une commande *STORE (*variable élément*)* a pour effet d'associer l'élément à la variable; cette dernière peut ensuite être utilisée en lieu et place de l'élément. Une commande *SUM (*liste_ouverte*)* est réducible à un élément de valeur égale à la somme des valeurs des éléments de la liste ouverte concernée; les commandes *PROD, MIN* et *MAX* ont des interprétations analogues. Une commande *DIFF* (élément élément*)* est réducible à un élément de valeur égale à

celle de la différence des éléments concernés; une commande *QUOT* est analogue. On remarque que les réductions ont eu lieu jusque dans les sous-listes.
En fin d'exécution, le système imprime une liste de l'état des variables.

```
***Etat final des variables***
   a: nombre(  1941.0000000000)
   x: nombre(  1.0000712529837)
   p: nombre( +.57277330308252822&+32)
   den: nombre(   .9999643747773)
   d: nombre( +.35625222657641610&-04)
   liste: sous-liste
      ***Impression d'une liste***
        1: nombre(  1941.0000000000)
        2: nombre(   54483870.00000)
        3: nombre(  -54481929.00000)
        4: nombre( +.35625222657641610&-04)
        5: nombre(  -54481929.00000)
        6: nombre(   54483870.00000)
        7: sous-liste
        ***Impression d'une liste***
          1: nombre(   94.00000000000)
          2: nombre( +.29683863369852300&+16)
          3: nombre( +.33688337246953581&-15)
          4: nombre(  1.0000712529837)
        <<<FIN>>>
        8: nombre(  1.0000356252227)
        9: nombre(    .9999643747773)
      <<<FIN>>>
   min: nombre(  -54481929.00000)
   s: nombre( +.29683863369911500&+16)
   g: nombre(   5823.000035625)
   c: nombre(  -54481929.00000)
   h: nombre( +.29683863369852300&+16)
   Une_liste_non_reductible: sous-liste
      ***Impression d'une liste***
        1: mot(premier_element)
        2: mot(deuxieme_element)
        3: mot(vient_ensuite_une_sous_liste)
        4: sous-liste
        ***Impression d'une liste***
          1: mot(grand-papa)
          2: mot(grand'maman)
        <<<FIN>>>
        5: mot(et_une_autre)
        6: sous-liste
        ***Impression d'une liste***
          1: entier(127)
          2: entier(32)
          3: entier(567)
          4: nombre(   569.0000000000)
          5: mot(ici_ca_s'imbrique)
          6: sous-liste
          ***Impression d'une liste***
            1: sous-liste
            ***Impression d'une liste***
            <<<FIN>>>
            2: sous-liste
            ***Impression d'une liste***
            <<<FIN>>>
            3: sous-liste
```

```
            ***Impression d'une liste***
              1: sous-liste
              ***Impression d'une liste***
              <<<FIN>>>
            <<<FIN>>>
            4: sous-liste
            ***Impression d'une liste***
              1: sous-liste
              ***Impression d'une liste***
                1: sous-liste
                ***Impression d'une liste***
                  1: nombre(  1941.0000000000)
                  2: entier(1)
                <<<FIN>>>
              <<<FIN>>>
            <<<FIN>>>
          <<<FIN>>>
        <<<FIN>>>
        7: mot(un_element_ordinaire)
        8: sous-liste
        ***Impression d'une liste***
        <<<FIN>>>
        9: mot(puis_le_dernier)
      <<<FIN>>>
   max: nombre( +.29683863369852300&+16)
   num: nombre(  1.0000356252227)
   b: nombre(   54483870.00000)
   u: nombre(   94.00000000000)
   zzz: entier(567)
   i: nombre( +.33688337246953581&-15)
   e: nombre(  -54481929.00000)
   f: nombre(   54483870.00000)
<<<FIN>>>
```

Pour avoir un système de traitement de liste complet, tel que le langage Lisp, il serait nécessaire de permettre de définir de nouvelles commandes, c'est-à-dire de nouvelles fonctions et permettre de les appliquer à des données. L'interprète *minilisp* ne possède pas cette possibilité. Sa structure générale est donnée dans la figure 31.

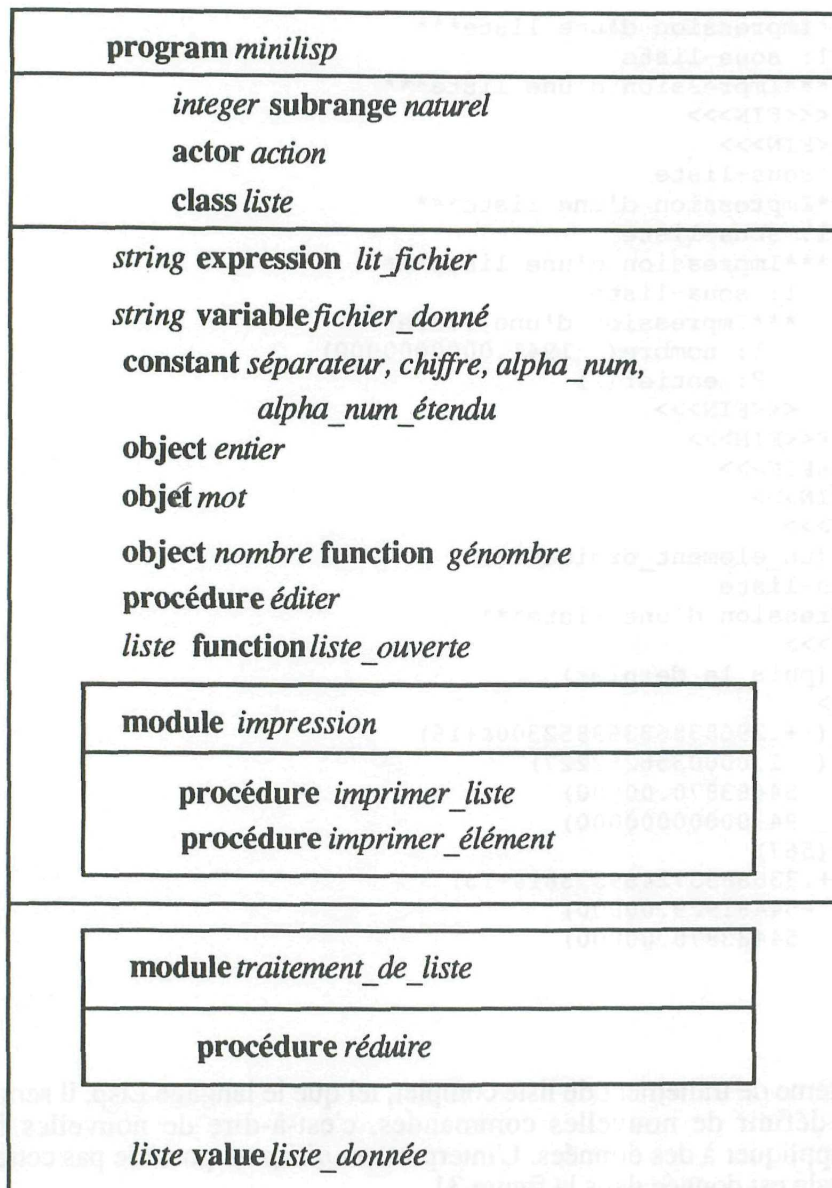


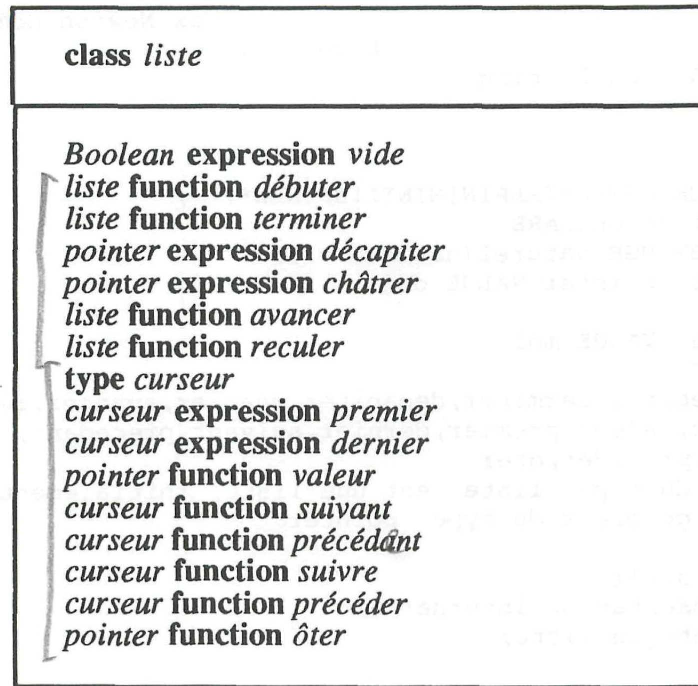
Figure 31

Dans un premier temps, le programme *minilisp* lit le fichier de données et le constitue en une seule chaîne stockée dans la variable *fichier_donné*. Dans cette chaîne, les lignes, du fichier original sont séparées au moyen du caractère **cr_char**; de plus, un caractère **ff_char** est appondu à la fin de la chaîne.

Au moyen de la fonction *liste_ouverte*, le fichier est ensuite analysé et structuré en une liste *liste_donnée* du type *liste*. Pour autant que le fichier ait en une forme correcte, cette liste est ensuite réduite au moyen de la procédure *réduire* du module *traitement_de_liste*.

La classe *liste* permet de construire des listes d'objets du type objet prédéfini généralisé *pointer*. Tout objet résultant d'une déclaration d'objet, de classe ou de processus peut être assimilé à une valeur du type *pointer* (on notera que ce n'est par contre pas le cas des rangées, tables, piles, queues ou objets procéduraux); le forçage inverse d'une valeur *obj* du type *pointer* en un objet d'un type résultant d'une déclaration d'objet , de classe ou de processus est possible: il y aura cependant une erreur à l'exécution du programme si *obj* n'est pas un objet du type approprié (sauf si *objet* = **nil**).

La structure de la classe *liste*, telle qu'elle est vue par l'utilisateur, est donnée dans la figure 32.



class *liste*

Boolean **expression** *vide*
liste **function** *débuter*
liste **function** *terminer*
pointer **expression** *décapiter*
pointer **expression** *châtrer*
liste **function** *avancer*
liste **function** *reculer*
type *curseur*
curseur **expression** *premier*
curseur **expression** *dernier*
pointer **function** *valeur*
curseur **function** *suivant*
curseur **function** *précédant*
curseur **function** *suivre*
curseur **function** *précéder*
pointer **function** *ôter*

Figure 32

Cette classe permet de gérer des listes de nature symétrique; ainsi, de nombreuses primitives apparaissent par paires liées aux deux sens de parcours possibles. C'est le cas des constructeurs *débuter* et *terminer*, des destructeurs *décapiter* et *châtrer*, des itérateurs *avancer* et *reculer*, des sélecteurs *premier* et *dernier* d'une part et *suivant* et *précédant* d'autre part ainsi que des constructeurs *suivre* et *précéder*.

La représentation interne des listes intervient au moyen du type objet *cercle* ; on reconnait la primitive de manipulation *nouer*.

```
minilisp                                          Vax Newton Compiler 0.2c5
                                                  Page     1
                       Source listing

/* /*OLDSOURCE=USER2:[RAPIN]MINILISP.NEW*/ */
PROGRAM minilisp DECLARE
  integer SUBRANGE naturel(naturel>=0);
  ACTOR action(pointer VALUE obj);

  CLASS liste  VALUE moi
    ATTRIBUTE
      vide,debuter,terminer,decapiter,chatrer,avancer,reculer,
      curseur,valeur,premier,dernier,suivant,precedant,
      suivre,preceder,oter
  (*Un objet du type  liste  est une liste, initialement vide,
    d'objets generaux du type  pointer
  *)
  DECLARE(*liste*)
    (***Representation interne***)
    OBJECT tete_de_liste;

    OBJECT cercle VALUE moi
      (pointer VALUE elt; cercle VARIABLE prec,suiv)
     VALUE tete=cercle(:tete_de_liste,moi,moi:);

    PROCEDURE nouer(cercle VALUE p,q) DO
      (p.suiv:=q.prec.suiv).prec:=q.prec
    DONE(*nouer*);

    (***Operations predefinies***)
    Boolean EXPRESSION vide=
    (*vrai ssi la liste est vide*)
      tete.suiv=tete;

    liste FUNCTION debuter
      (pointer VALUE obj)
    (*Insere l'objet  obj  en debut de la liste concernee  moi ;
      le resultat est la liste  moi .
    *)
    DO nouer(tete,cercle(:obj,moi,moi:)) TAKE moi DONE;

    liste FUNCTION terminer
      (pointer VALUE obj)
    (*Insere l'objet  obj  en fin de la liste concernee  moi ;
      le resultat est la liste  moi .
    *)
    DO nouer(cercle(:obj,moi,moi:),tete) TAKE moi DONE;

    pointer EXPRESSION decapiter=
    (*Elimine l'element au debut de la liste et retourne l'objet
      correspondant; en cas de liste vide, le resultat est NIL
      sans autre effet.
    *)
      CONNECT tete.suiv THEN
        TAKE INSPECT elt WHEN
        tete_de_liste THEN NIL
```

```
    DEFAULT nouer(prec,suiv) TAKE elt DONE
  DONE;

pointer EXPRESSION chatrer=
(*Elimine l'element en fin de la liste et retourne l'objet
  correspondant; en cas de liste vide, le resultat est NIL
  sans autre effet.
*)
  CONNECT tete.prec THEN
    TAKE INSPECT elt WHEN
    tete_de_liste THEN NIL
    DEFAULT nouer(prec,suiv) TAKE elt DONE
  DONE;

liste FUNCTION avancer(action VALUE acte)
(*Parcourt, du debut a la fin, la liste concernee; effectue
  l'enonce  acte[obj]  pour l'objet  obj  inclus dans chacun
  de ses elements. Le resultat est la liste concernee  moi .
*)
DECLARE cercle VARIABLE curs:=tete DO
  UNTIL (curs:=curs.suiv)=tete REPEAT
    acte[curs.elt]
  REPETITION
TAKE moi DONE(*avancer*);

liste FUNCTION reculer(action VALUE acte)
(*Parcourt, de la fin au debut, la liste concernee; effectue
  l'enonce  acte[obj]  pour l'objet  obj  inclus dans chacun
  de ses elements. Le resultat est la liste concernee  moi .
*)
DECLARE cercle VARIABLE curs:=tete DO
  UNTIL (curs:=curs.prec)=tete REPEAT
    acte[curs.elt]
  REPETITION
TAKE moi DONE(*reculer*);

TYPE curseur=cercle;
(*un curseur denote un element de la liste*)

curseur FUNCTION curs(curseur VALUE c) DO
  TAKE INSPECT c.elt WHEN
  tete_de_liste THEN NIL
  DEFAULT c DONE
DONE(*curs*);

curseur EXPRESSION premier=
(*Un curseur au premier element de la liste  moi ; si la liste
  est vide, le resultat est NIL.
*)
  curs(tete.suiv);

curseur EXPRESSION dernier=
```

```
(*Un curseur au dernier element de la liste  moi ; si la liste
  est vide, le resultat est NIL.
*)
  curs(tete.prec);

pointer FUNCTION valeur(curseur VALUE c)DO
(*L'objet inclus dans l'element designe par le curseur  c ;

  Condition d'emploi:  c~=NIL
*)
TAKE c.elt DONE;

curseur FUNCTION suivant(curseur VALUE c)DO
(*Un curseur a l'element suivant celui designe par  c ;
  si  c  est l'element en fin de liste, le resultat est NIL.

  Condition d'emploi:  c~=NIL
*)
TAKE curs(c.suiv) DONE;

curseur FUNCTION precedant(curseur VALUE c)DO
(*Un curseur a l'element precedant celui designe par  c ;
  si  c  est l'element en tete de liste, le resultat est NIL.

  Condition d'emploi:  c~=NIL
*)
TAKE curs(c.prec) DONE;

curseur FUNCTION suivre
   (curseur VALUE c; pointer VALUE obj)
(*Place, a la suite de l'element  c , un nouvel element contenant
  l'objet  obj . Le resultat est un curseur au nouvel element.

  Condition d'emploi:  c~=NIL
*)
DECLARE curseur VALUE n=cercle(:obj,moi,moi:) DO
  nouer(c,n)
TAKE n DONE(*suivre*);

curseur FUNCTION preceder
   (curseur VALUE c; pointer VALUE obj)
(*Place, avant l'element  c , un nouvel element contenant
  l'objet  obj . Le resultat est un curseur au nouvel element.

  Condition d'emploi:  c~=NIL
*)
DECLARE curseur VALUE n=cercle(:obj,moi,moi:) DO
  nouer(n,c)
TAKE n DONE(*preceder*);

pointer FUNCTION oter
  (curseur VALUE c)
```

```
minilisp                                          Vax Newton Compiler 0.2c5
                                             Page      4
                          Source listing

  490      (*Elimine de la liste l'element designe par le curseur  c ; le
  490        resultat est l'objet inclus dans l'element enleve.
  490
  490        Condition d'emploi:  c~=NIL
  490       *)
  490      DO nouer(c.prec,c.suiv) TAKE c.elt DONE(*oter*)
  506    DO(*liste*)DONE;
  509    /* /*EJECT*/ */
```

On remarque que l'on fait en sorte de toujours conserver un noeud *tête* dans la liste circulaire représentative: ce noeud inclut un objet *elt* du type *tête_de_liste*. La présence d'une telle tête de liste simplifie la réalisation de primitives telles que le constructeur *débuter* ou le destructeur *ôter*: il n'est pas nécessaire de traiter spécialement l'insertion du premier élément ou l'élimination du dernier élément de la liste.
On remarque le type *curseur* introduit au moyen de la déclaration **type** *curseur* = *cercle* . A l'intérieur de la classe *liste*, ce type est strictement équivalent au type *cercle*; le type *curseur* est un attribut de la classe *liste* mais non le type *cercle* : ceci implique que *curseur* est exporté de la classe comme type objet nu: ses attributs, donc le détail de la représentation interne de la liste, ne sont pas connus en dehors de la classe. Ceci implique qu'à l'extérieur de la classe, des valeurs du type *curseur* ne sont manipulables que par l'intermédiaire de primitives (telles que le sélecteur *premier*, le constructeur *suivre* ou le destructeur *ôter*) également exportées de la classe. On remarque que les valeurs du type *curseur* sont normalement produites par l'intermédiaire de la fonction *curs* qui fait en sorte de livrer un objet vide **nil** lorsque l'on fait avancer ou reculer un curseur au-delà de l'extrémité de la liste.

Ce programme illustre à plusieurs reprises la manière de traiter les objets généraux du type *pointer* au moyen d'énoncés **inspect**. D'une manière générale, une clause **inspect** est considérée comme une variante; une telle variante a la forme:

> **inspect** expression **when**
> suite_d_inspections

La suite d'inspections comporte une ou plusieurs inspections séparées par le symbole | ; chaque inspection a la forme:

> type_objet **value** identificateur alternative

L'expression entre **inspect** et **when** doit livrer un objet éventuellement vide, du type *pointer* ou réducible à ce type.

Une variante **inspect** est satisfaite ssi l'expression livre un objet non vide d'un type préfixant l'une des inspections. Le cas échéant, l'alternative contenue dans l'inspection correspondante est élaborée; pendant son exécution, l'objet sur lequel porte la variante est connecté: ses attibuts sont disponibles sans qualifications explicite. La clause **value** identificateur entre le nom du type objet préfixant une inspection et l'alternative correspondante est facultative; le cas échéant, l'identificateur correspondant dénote l'objet sur lequel porte la variante sous son type original (et non sous le type *pointer*). Au moyen de cet identificateur de valeur, il est possible d'utiliser directement une valeur du type *pointer* pour atteindre les attributs d'un objet ou l'indicer.

Dans la suite du programme, la fontion *liste_ouverte* consomme la partie initiale de la chaîne contenue dans la variable repérée par le paramètre *texte* ayant la forme d'une liste ouverte; elle construit comme résultat un objet du type *liste* . Les éléments de la liste ouverte donnée y sont représentés au moyen d'objets *entier*, *mot* ou *liste*; des objets du type *nombre* n'y sont insérés qu'après réduction de la liste. On constate que le recours aux objets du type *pointer* permet de créer des structures de données dont les éléments sont inhomogènes; il n'y a aucune difficulté particulière à créer des listes dont certains éléments sont des sous-listes du même type *liste*.

Dans le module subséquent *impression*, il a été implanté les procédures mutuellement récursives *imprimer_liste* et *imprimer_élément* au moyen d'objets acteurs selon la technique expliquée au chapitre 5.
Du module *traitement_de_liste* est exportée la procédure *réduire* utilisée pour réduire une liste *lis* fournie en paramètre en fonction des commandes que cette liste contient. Ce module a la structure décrite à la figure 33.

```
string EXPRESSION lit_fichier=
(*Lecture du fichier donne et compaction de ce dernier en une
  chaine
*)
  DECLARE
    string VARIABLE texte:="",ligne; integer VARIABLE num:=0
  DO
    print("*****Listage fichier source*****");
    UNTIL end_file REPEAT
      read(ligne);
      texte:=texte+ligne+CR_CHAR;
      line; edit((num:=SUCC num),5,0); print("--->",ligne)
    REPETITION;
    print(line,"<<<FIN DU FICHIER>>>")
  TAKE texte+FF_CHAR DONE;

string VARIABLE fichier_donne:=lit_fichier;

CONSTANT separateur={" ",CR_CHAR},
         chiffre={#0123456789#},
         alpha_num={FROM "A" TO "Z",FROM "a" TO "z",#0123456789#},
         alpha_num_etendu={FROM "A" TO "Z",FROM "a" TO "z",
                           #@$&0123456789-_.'#};
OBJECT entier(string VALUE valeur);
OBJECT mot(string VALUE texte);
OBJECT nombre(real VALUE valeur)
  FUNCTION genombre(entier VALUE ent)
DECLARE real VARIABLE x:=0 DO
  THROUGH ent.valeur VALUE dig REPEAT
    x:=10*x+(ORD dig-ORD "0")
  REPETITION
TAKE nombre(x) DONE;

PROCEDURE editer
   (real VALUE x)
DECLARE real VALUE abs_x=ABS x DO
  UNLESS FINITE x THEN
    print(x) ELSE
  IF x=0 THEN
    print(_".0") ELSE
  IF abs_x>integer MAX\/abs_x<.1 THEN
    print(x)
  DEFAULT
    edit(x,17,13-ln(abs_x)/ln(10))
  DONE
DONE(*editer*);

liste FUNCTION liste_ouverte
   (string REFERENCE texte)
(*Analyse la chaine  texte ; en elimine la partie initiale
   ayant la forme d'une liste ouverte:

```

```
783     liste_ouverte =[separateurs][element]
783                    {separateurs element}
783                    [separateurs];
783     separateurs   =separateur{separateur};
783     element       =atome|liste;
783     atome         =mot|entier;
783     entier        =chiffre{chiffre};
783     mot           =identificateur-entier;
783     identificateur=alpha_num[{alpha_num_etendu} alpha_num];
783     liste         ="("liste_ouverte")".
783   *)
783   DECLARE(*liste_ouverte*)
784     liste VALUE lo=liste;
790     string VARIABLE nou_texte,identificateur
795   DO(*liste_ouverte*)
796     CYCLE traite_elements REPEAT
799       IF "(" STARTS (texte:=separateur-texte) THEN
810         nou_texte:=texte LEFTCUT "(";
816         lo.terminer(liste_ouverte(nou_texte));
826         IF ")" STARTS nou_texte THEN
831           texte:=nou_texte LEFTCUT ")"
836         DEFAULT
837           lo.chatrer
840         EXIT traite_elements DONE ELSE
844       IF alpha_num STARTS texte THEN
849         identificateur:=
851           (alpha_num_etendu SPAN texte)TORIGHT alpha_num;
859         texte:=texte LEFTCUT identificateur;
865         lo.terminer
868           (IF identificateur IN chiffre THEN
874              entier(identificateur)
878            DEFAULT mot(identificateur) DONE)
885       DEFAULT EXIT traite_elements DONE
889     REPETITION
890   TAKE lo DONE(*liste_ouverte*);
894   /* /*EJECT*/ */
```

```
minilisp                                          Vax Newton Compiler 0.2c5
                                           Page     7
                     Source listing

MODULE impression
  ATTRIBUTE imprimer_liste, imprimer_element
DECLARE(*impression*)
  ACTOR acteur_imprimer_liste(liste VALUE lis; naturel VALUE niv)
    VARIABLE impression_liste;
  ACTOR acteur_imprimer_element(pointer VALUE obj; naturel VALUE niv)
    VARIABLE impression_element;

  PROCEDURE imprimer_liste
     (liste VALUE lis; naturel VALUE niv)
  DECLARE(*imprimer_liste*)
    integer VARIABLE num:=0;

    PROCEDURE ligne DO line; column(1+2*niv) DONE
  DO(*imprimer_liste*)
    print(ligne,"***Impression d'une liste***");
    lis.avancer
      (BODY action(pointer VALUE obj) DO
         print(ligne,edit((num:=SUCC num),3,0),":"_);
         impression_element[obj,niv];
       DONE);
    print(ligne,"<<<FIN>>>")
  DONE(*imprimer_liste*);

  PROCEDURE imprimer_element
     (pointer VALUE obj; naturel VALUE niv)
  DO(*imprimer_element*)
    INSPECT obj WHEN
    liste VALUE lis THEN
      print("sous-liste");
      impression_liste[lis,SUCC niv]|
    entier THEN print("entier(",valeur,")")|
    mot THEN print("mot(",texte,")")|
    nombre THEN print("nombre(",editer(valeur),")")
    DEFAULT print("###element inconnu###") DONE
  DONE(*imprimer_element*)
DO(*impression*)
  impression_liste:=acteur_imprimer_liste(imprimer_liste);
  impression_element:=acteur_imprimer_element(imprimer_element)
DONE(*impression*);
/* /*EJECT*/ */
```

Dans ce module, la variable *réduction* a été introduite à cause de la récursion mutuelle entre la procédure *réduire* et plusieurs procédures du module.

Il est ensuite défini une table extensible *var* des variables recontrées lors des réductions de listes; lorsque la chose devient nécessaire, l'extension intervient au moyen de la procédure *restructure_var_table*.

module *traitement_de_liste*
liste **function** *transforme_liste* *transforme_liste* **variable** *réduction*
constant *capac_init, rempli_min, rempli_max* *pointer* **table** *obj_table* *obj_table* **variable** *var* **procedure** *restructure_var_table*
real **function** *évaluer*
constant *max_foncs* *pointer* **functor** *opérateur_liste* *opérateur_liste* **table** *oper_table* *oper_table* **value** *foncs* *real* **functor** *dyad*
pointer **function** *val* *opérateur_liste* **function** *itère_dyad* *opérateur_liste* **function** *applique_dyad*
pointer **function** *store*
liste **function** *réduire*

Figure 33

La fonction récursive *évaluer* n'est applicable qu'à un élément de liste susceptible de produire une valeur numérique. Dans le cas où cet élément est une variable, *évaluer* est appliquée récursivement à son contenu. Dans le cas où l'élément est une liste, celle-ci est d'abord réduite: il y a erreur si elle ne peut être réduite à un seul élément; dans la cas contraire, *évaluer* est appliqué récursivement à cet élément unique.

```
MODULE traitement_de_liste
  ATTRIBUTE reduire
DECLARE(*traitement_de_liste*)
  liste FUNCTOR transforme_liste(liste VALUE lis)
    VARIABLE reduction;

  CONSTANT capac_init=20,rempli_min=.5,rempli_max=.8;
  pointer TABLE obj_table VARIABLE var:=obj_table(capac_init);
  PROCEDURE restructure_var_table DO
    THROUGH
      (obj_table(CARD var%rempli_min)=:var)
      INDEX nom  VALUE obj
    REPEAT var[nom]:=obj REPETITION
  DONE(*restructure_var_table*);

  real FUNCTION evaluer
    (pointer VALUE obj)
  DO(*evaluer*)TAKE
    INSPECT obj WHEN
    liste VALUE terme THEN
      IF TAKE
        UNLESS
          reduction[terme].vide
        THEN suivant(premier)~=NIL DEFAULT TRUE DONE
      THEN
        print(line,"###EVAL ne porte pas sur liste reductible"_
                   "a un element###");
        imprimer_liste(terme,0)
      RETURN DONE
    TAKE evaluer(valeur(premier))|

    entier VALUE ent THEN genombre(ent).valeur|
    nombre THEN valeur|
    mot THEN
      UNLESS var ENTRY texte THEN
        print(line,"###EVAL porte sur variable `",texte,
                   "' non initialisee###")
      RETURN DONE
    TAKE evaluer(var[texte]) DONE
  DONE(*evaluer*);

  CONSTANT max_foncs=20;
  pointer FUNCTOR operateur_liste(liste VALUE lis)
    TABLE oper_table VALUE foncs=oper_table(max_foncs);
  real FUNCTOR dyad(real VALUE x,y);

  pointer FUNCTION val
    (pointer VARIABLE p)
  DO
    CYCLE examen DO
      INSPECT p WHEN
      mot THEN
```

```
            IF var ENTRY texte THEN
              p:=var[texte]
            REPEAT examen DONE
          DONE(*INSPECT p*)
        DONE(*CYCLE examen*)
      TAKE p DONE(*val*);

     operateur_liste FUNCTION itere_dyad
       (real VALUE init; dyad VALUE op)
     DECLARE(*itere_dyad*)
       operateur_liste VALUE iteration=
         BODY
           operateur_liste(liste VALUE lis)
         DECLARE real VARIABLE r:=init DO
           CONNECT reduction[lis] THEN
             DECLARE
               curseur VARIABLE c:=premier;
               pointer VARIABLE v
             DO
               UNTIL c=NIL REPEAT
                 r:=op[r,evaluer(INSPECT v:=val(valeur(c)) WHEN
                                 liste VALUE sous_liste THEN
                                   iteration[sous_liste]
                                 DEFAULT v DONE)];
                 c:=suivant(c)
               REPETITION
             DONE(*DECLARE c*)
           DONE(*CONNECT reduction[lis]*)
         TAKE nombre(r) DONE
     DO(*itere_dyad*)TAKE
       iteration
     DONE(*itere_dyad*);

     operateur_liste FUNCTION appliq_dyad
       (dyad VALUE op)
     DO(*appliq_dyad*)TAKE
       BODY
         operateur_liste(liste VALUE lis)
       DO
         TAKE CONNECT reduction[lis] THEN
           TAKE DECLARE curseur VALUE p=premier,d=dernier DO
             IF TAKE
               UNLESS lis.vide THEN
                 suivant(p)~=d
               DEFAULT TRUE DONE
             THEN
               print(line,"###Operation dyadique ne porte pas sur"_
                          "liste a deux elements###");
               imprimer_liste(lis,0)
             RETURN DONE
           TAKE
             nombre(op[evaluer(valeur(p)),evaluer(valeur(d))])
```

```
         DONE(*TAKE DECLARE p,d*)
       DONE(*TAKE CONNECT reduction[lis]*)
     DONE(*BODY operateur_liste*)
   DONE(*appliq_dyad*);

   pointer FUNCTION store
     (liste VALUE lis)
   DECLARE(*store*)
     pointer VALUE val_premier=lis.decapiter;
     pointer VARIABLE val_dernier
   DO(*store*)TAKE
     CONNECT lis THEN
       TAKE CYCLE stockage DO
         UNLESS reduction[lis].vide THEN
           val_dernier:=chatrer;
           IF vide THEN
             INSPECT val_premier WHEN
             mot THEN
               IF CARD var>=rempli_max*CAPACITY var THEN
                 restructure_var_table
               DONE(*IF CARD var>=rempli_max*CAPACITY var*)
             EXIT stockage TAKE (var[texte]:=val_dernier) DONE
           DONE(*IF vide*);
           terminer(val_dernier)
         DONE(*UNLESS reduction[lis].vide*);
         print(line,"###Liste ne denote pas un stockage###");
         imprimer_liste(debuter(val_premier),0)
       RETURN DONE(*TAKE CYCLE stockage*)
     DONE(*CONNECT lis*)
   DONE(*store*);

   liste FUNCTION reduire
     (liste VALUE lis)
   DO(*reduire*)
     CONNECT lis THEN
       DECLARE curseur VARIABLE c:=premier DO
         UNTIL c=NIL REPEAT
           CYCLE discussion DO
             INSPECT valeur(c) WHEN
             mot THEN
               IF var ENTRY texte THEN
                 c:=suivre(c,var[texte]); oter(precedant(c))
               REPEAT discussion DONE;

               IF foncs ENTRY texte THEN
                 INSPECT val(valeur(suivant(c))) WHEN
                 liste VALUE sous_lis THEN
                   c:=lis.suivre(c,foncs[texte][sous_lis]);
                   lis.oter(lis.precedant(c));
                   lis.oter(lis.suivant(c))
                 DONE(*INSPECT val(valeur(suivant(c)))*)
               DONE(*IF foncs ENTRY texte*)|
```

```
minilisp                                        Vax Newton Compiler 0.2c5
                                          Page   11
                     Source listing

                  liste VALUE sous_liste THEN reduire(sous_liste)
                  DONE(*INSPECT valeur(c)*)
                DONE(*CYCLE discussion*);
                c:=suivant(c)
              REPETITION(*UNTIL c=NIL*)
            DONE(*DECLARE c*)
          DONE(*CONNECT lis*)
        TAKE lis DONE(*reduire*)
      BEGIN(*traitement_de_liste*)
        reduction:=transforme_liste(reduire);
        foncs["MIN"]:=itere_dyad(INFINITY,BODY dyad DO TAKE x MIN y DONE);
        foncs["MAX"]:=itere_dyad(-INFINITY,BODY dyad DO TAKE x MAX y DONE);
        foncs["SUM"]:=itere_dyad(0,BODY dyad DO TAKE x+y DONE);
        foncs["PROD"]:=itere_dyad(1,BODY dyad DO TAKE x*y DONE);
        foncs["DIFF"]:=appliq_dyad(BODY dyad DO TAKE x-y DONE);
        foncs["QUOT"]:=appliq_dyad(BODY dyad DO TAKE x/y DONE);
        foncs["STORE"]:=operateur_liste(store)
      INNER(*traitement_liste*)
        print(page,"***Etat final des variables***");
        THROUGH var  INDEX ident  VALUE obj  REPEAT
          print(line,___,ident,":"_,imprimer_element(obj,2))
        REPETITION;
        print(line,"<<<FIN>>>")
      END(*traitement_liste*);
      /* /*EJECT*/ */
```

La table *oper_table* est la table des commandes enregistrées sous la forme d'objets foncteurs du type *opérateur_liste*.

La fonction *val* remplace un élément de liste dénotant une variable par le contenu de cette dernière; appliqué à une autre forme, l'élément, elle laisse ce dernier inchangé.

La fonction *intère_dyad* produit les objets foncteurs nécessaires à l'interprétation des commandes *SUM, PROD, MAX et MIN*. Le lecteur peut constater la manière dont on s'y est pris pour que l'algorithme résultant soit applicable récursivement à des éléments de la forme d'une sous-liste.

Dans cette primitive, ainsi que plusieurs autres de ce module, on a fait intervenir une variable du type *curseur*; cette variable *c* est utilisée pour parcourir la liste *lis* passée en paramètre. On notera à ce propos qu'un tel type exporté comme attibut d'une classe ne peut être utilisé pour définir des entités de ce type que dans un bloc incorporé dans une région (variante **connect** , inspection) dans laquelle un objet de cette classe est connecté. En particulier, il n'est pas autorisé de recourir à une qualification explicité pour définir une telle entité: ainsi, la déclaration *lis.curseur* **variable** *c* est incorrecte.

La fonction *applique_dyad* produit les objets nécessaires à l'interprétation des commandes *DIFF* et *QUOT* tandis que la commande *STORE* est interprétée par l'intermédiaire de la fonction *store*.

Dans la fonction *réduire*, on remarque la manière dont sont traitées les commandes. Incorporé au test **if** *foncx* **entry** *texte*, on examine si la commande est suivie d'une liste au moyen de l'énoncé **inspect** *val (valeur (suivant (c)))*; si c'est bien le cas, l'inspection débutant par la clause *liste* **value** *sous_liste* est satisfaite.

A l'intérieur de l'aternative subséquente, on remarque plusieurs opérations sur la liste *lis*; celles-ci ont dû être qualitfiées explicitement: dans le cas contraire, ces opérations porteraient sur la sous-liste du fait de la connection impliquée par l'inspection. Aussi, l'énoncé *c := suivre (c, foncs [teste] [sous-lis])* signifierait *c := sous_lis.suivre [texte] [sous-lis]).*

La table des commandes *foncs* est initialisée dans le prologue du module *traitement_de_liste* tandis que épilogue fait imprimer l'état final des variables.

Le reste du programme ne nécessite pas de commentaire particulier; il est donné sans autres.

```
minilisp                                     Vax Newton Compiler 0.2c5
                                         Page   12
                   Source listing


     liste VALUE liste_donnee=liste_ouverte(fichier_donne)
   DO(*minilisp*)
     print(page,"***Liste obtenue***");
     imprimer_liste(liste_donnee,0);
     IF FF_CHAR STARTS fichier_donne THEN
       print(line,"<<<FICHIER DONNE A ETE ENTIEREMENT TRAITE>>>");
       print(page,"***Liste transformee***");
       imprimer_liste(reduire(liste_donnee),0)
     DEFAULT
       print(page,"###Fichier donne non epuise; partie non traitee###");
       UNTIL
         print(line,fichier_donne LEFTOF CR_CHAR)
       TAKE
         FF_CHAR STARTS (fichier_donne:=fichier_donne LEFTCUT CR_CHAR)
       REPETITION;
       print(line,"###FIN DU FICHIER###")
     DONE
   DONE(*minilisp*)

      ****  No messages were issued  ****
```

Chapitre 11

Le retour arrière

Les algorithmes de nature le plus classique sont de nature déterministe, dans le sens qu'à chaque étape la décision à prendre est connue à priori. En pratique, il n'en va cependant pas toujours ainsi. Par exemple, si l'on veut réaliser un programme capable de trouver le chemin menant au but dans un labyrinthe, il n'est pas possible de décider à priori quelle bifurcation prendre lorsque l'on parvient à un carrefour; un tel programme ne sera donc pas déterministe.

C'est au moyen de la technique du retour arrière (en anglais "back-tracking") que l'on peut programmer ce genre d'application non déterministe. L'idée est simple; on va en exposer le principe dans le cas du labyrinthe. On programme un parcours de ce dernier. A chaque bifurcation, on fait un choix (à priori arbitraire) du prochain couloir à suivre, tout en se rappelant d'où l'on est venu et les choix déjà effectués. Le labyrinthe étant par hypothèse fini, on aboutira au bout d'un temps fini à l'une des situations suivantes:

- On a atteint le but: le chemin parcouru est donc le bon.

- On aboutit à une impasse: on revient en arrière au dernier carrefour duquel est issu un chemin non encore exploré. A cette bifurcation, on choisit un couloir non parcouru et on reprend l'exploration à partir de ce dernier. C'est-là le cas du retour arrière.

- On aboutit à un carrefour déjà visité: le labyrinthe possède un cycle. Dans ce cas aussi, on fait un retour arrière; au préalable, il faudra évidemment faire en sorte que le couloir par lequel on est parvenu à cette bifurcation soit considéré comme déjà exploré.

- Le retour arrière aboutit à l'origine du parcours dont il n'est issu aucun chemin non exploré: le but est dans ce cas inatteignable. Cette circonstance ne peut intervenir que lorsque tous les couloirs atteignables depuis l'origine ont été explorés.

En résumé, en un temps fini on aura soit trouvé un chemin menant au but soit prouvé que ce dernier ne peut être atteint.

Les applications que l'on rencontre en intelligence artificielle sont souvent de nature non déterministe. Pour cette raison, dans le langage de programmation Prolog (essentiellement orienté vers les applications de l'intelligence artificielle), la structure de contrôle principale est le retour arrière (au même titre que la procédure récursive peut être considérée comme la structure de contrôle principale des langages algorithmiques "classiques" tels que Pascal ou Newton).

En pratique, le retour arrière peut être programmé à l'aide d'une pile de coroutines, appelée la traînée. Dans le cas du labyrinthe, la traînée modélise le fil d'Arianne. Dans la traînée, on empile à chaque point de décision une coroutine chargée de produire, chaque fois qu'elle est attachée, un choix non encore essayé et de terminer son exécution lorsqu'il n'en reste plus.

Un inconvénient du retour arrière est que cette technique implique souvent un volume de calcul considérable. On peut en-effet admettre que ce volume croit exponentiellement par rapport au nombre de points de décision; il y a donc risque d'explosion combinatoire. Il est parfois possible, voire nécessaire, de remédier à cette explosion combinatoire en adoptant une heuristique.
Aux points de décision, au lieu de choisir arbitrairement l'ordre dans lequel seront explorées les différentes variantes possible, ces dernières seront ordonnées selon certains critères; ceux-ci dépendront de l'application spécifique. Ainsi, il peut arriver que l'on sache que certains choix ont plus de chances d'aboutir à une solution que d'autres; on pourra donc avoir avantage à les explorer en premier. Dans d'autres cas, il est possible de discerner rapidement si une variante donnée aboutit à une solution ou à une impasse; il y a tout intérêt à l'essayer en priorité. Si cette variante conduit à une solution, celle-ci sera rapidement trouvée; si, par contre, elle aboutit à une

impasse, le temps perdu pour le prouver ne sera pas prohibitif. En règle générale, il n'est cependant pas facile de choisir de bons critères heuristiques. Dans certains cas, par exemple dans les programmes de jeu d'échec, l'introduction d'une bonne heuristique ne permet pas d'éviter l'explosion combinatoire; il est alors nécessaire de se limiter à une exploration partielle. Des critères heuristiques sont alors utilisés non seulement pour ordonner les choix envisageables, mais pour éliminer certaines variantes dont le succès apparait très improbable; de plus, les variantes retenues ne sont explorées qu'à une longueur (un nombre de points de décision) relativement faible: leur valeur est ensuite estimée au moyen d'une heuristique (on fonction d'évaluation) ad-hoc.

On va d'abord montrer la manière de réaliser certains algorithmes d'analyse syntaxique au moyen du retour arrière. L'interprète d'expressions arithmétiques présenté au chapitre 9 est basé sur la méthode de la descente récursive. Cette dernière méthode n'est applicable qu'à des grammaires satisfaisant à des restrictions relativement sévères.

A titre d'exemple, on va réaliser des algorithmes d'analyse syntaxique pour la grammaire suivante:

```
G ( T { x, y, z, f, g, h, +, -, *, /, ^, (,) },
    N { E, A, T, M, F, B, V, D},
    E,
    R { E → T ; ( E → ( AT; ( E → (EAT;
        A → + ; A → - ;
        T → F ; ( T → ( TMF ;
        M → * ; M → / ;
        F → B ; F) → B ^ F ) ; FA → B ^ FA ; FM → B ^ FM;
        B → V ; B → D ( E ) ; B → ( E ) ;
        V → x ; V → y; V → z ;
        D → f ; D → g ; D → h } )
```

Les membres gauches de certaines règles de production ont plus d'un symbole; cette grammaire n'est pas indépendante du contexte: elle ne peut être analysée par la méthode de la descente récursive. On constate par contre que le membre droit de chaque règle de production est au moins aussi long que son membre gauche. Il s'ensuit que la grammaire peut être analysée au moyen d'un algorithme incorporant la technique du retour arrière.

D'une manière générale, on considère deux catégories d'algorithmes d'analyse syntaxique.

- Un algorithme ascendant part du mot à analyser. Il cherche à transformer ce mot par substitutions du membre gauche de certaines règles de production au membre droit correspondant jusqu'à l'obtention de l'axiome de la grammaire. Chaque substitution est appelée une réduction.

- Un algorithme descendant part de l'axiome de la grammaire. Il cherche à le transformer au moyen d'une suite de substitutions du membre droit de certaines règles de production au membre gauche correspondant jusqu'à l'obtention du mot à analyser. Chaque substitution est appelée une expansion.

Dans le cas de la grammaire proposée, les deux approches sont applicables. On va tout d'abord présenter un programme *ascendant* incorporant un algorithme d'analyse de nature ascendante.

Ce programme a pour effet de lire un fichier de données au moyen duquel sont introduits les mots à analyser. Ceux-ci y sont séparés par n'importe quelle chaîne non vide de caractères non terminaux ou par le passage d'une ligne du fichier à la suivante. La structure du programme est donnée à la figure 34.

Dans ce programme, le module *grammaire* inclut de manière très naturelle la grammaire à analyser. Les alphabets *terminal* et *non_terminal* définissent les ensembles de symboles terminaux et auxiliaires respectivement. L'axiome y apparait sous la forme d'un caractère et les règles de production sous celle d'une (longue) chaîne dans laquelle le membre gauche de chaque règle est suivie de la paire → et le membre droit d'un espace. Il serait manifestement possible d'analyser d'autre grammaires en faisant des changements relativement évidents à ce module.

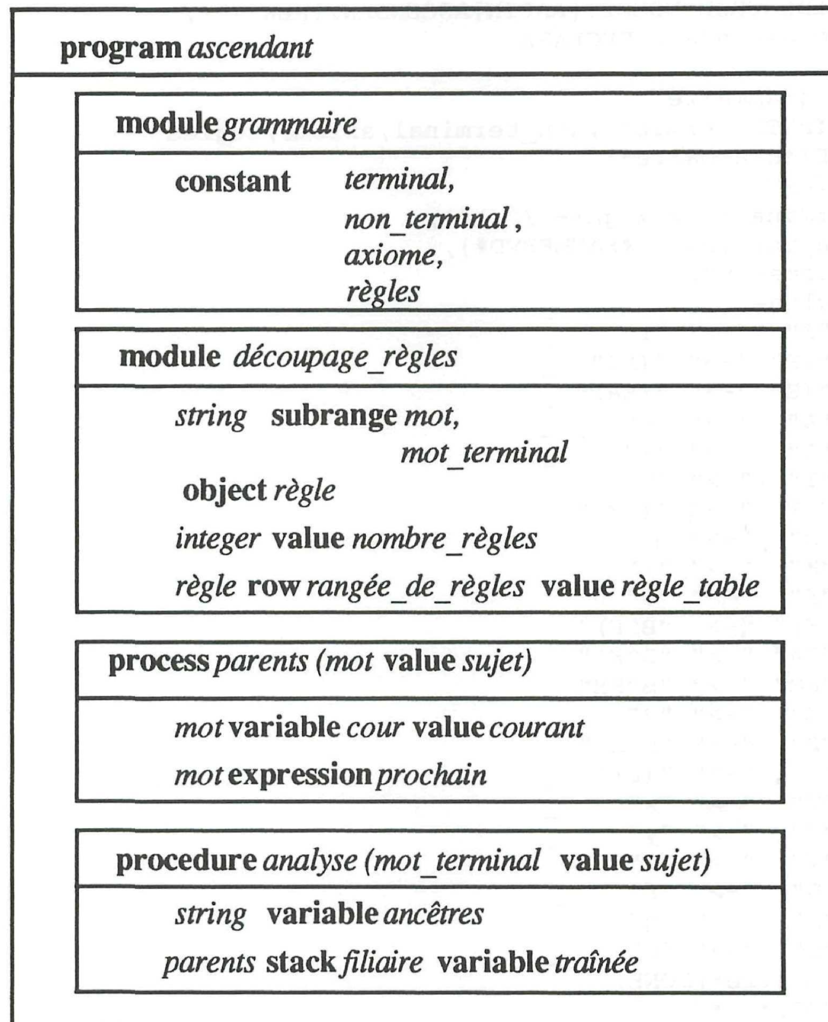


Figure 34

Dans le module *découpage_règles*, il est établi une autre représentation de la suite de règles de production au moyen de la rangée *règle_table* d'objets du type *règle* dont les deux composantes *gauche* et *droite* sont les membres d'une des règles de production. Dans cette rangée, les règles sont triées selon les critères suivants.

```
ascendant                                    Vax Newton Compiler 0.2c9
                                        Page    1
                  Source listing

/* /*OLDSOURCE=USER2:[RAPIN]ASCENDANT.NEW*/ */
PROGRAM ascendant DECLARE

MODULE grammaire
  ATTRIBUTE terminal,non_terminal,axiome,regles
DECLARE(*grammaire*)
  CONSTANT
    terminal={#xyzfgh+-*/^()#},
    non_terminal={#EATMFBVD#},
    axiome="E",
    regles=                    _
      "E"  "->" "T"            _
      "(E" "->" "(AT"          _
      "(E" "->" "(EAT"         _
      "A"  "->" "+"            _
      "A"  "->" "-"            _
      "T"  "->" "F"            _
      "(T" "->" "(TMF"         _
      "M"  "->" "*"            _
      "M"  "->" "/"            _
      "F"  "->" "B"            _
      "F)" "->" "B^F)"         _
      "FA" "->" "B^FA"         _
      "FM" "->" "B^FM"         _
      "B"  "->" "V"            _
      "B"  "->" "D(E)"         _
      "B"  "->" "(E)"          _
      "V"  "->" "x"            _
      "V"  "->" "y"            _
      "V"  "->" "z"            _
      "D"  "->" "f"            _
      "D"  "->" "g"            _
      "D"  "->" "h"            _
DO(*grammaire*)DONE;
/* /*EJECT*/ */
```

En priorité sont placées les règles dont la différence des longueurs entre le membre gauche et le membre droit est la plus grande: en-effet, ces règles, lorsqu'elles sont applicables, réduiront plus vite le mot: il est donc raisonnable de chercher à les appliquer en premier. A différence de longueur égale, les règles sont ordonnées par ordre croissant des longueurs de leurs membres.

```
MODULE decoupage_regles
  ATTRIBUTE mot,mot_terminal,regle,nombre_regles,regle_table
DECLARE(*decoupage_regles*)
  alphabet VALUE complet=terminal+non_terminal;
  string SUBRANGE mot(mot IN complet)
         SUBRANGE mot_terminal(mot_terminal IN terminal);
  mot SUBRANGE mot_non_terminal(mot_non_terminal<>non_terminal);

  integer VALUE nombre_regles=
    DECLARE
      string VARIABLE curs:=regles;
      integer VARIABLE compte:=0
    DO
      WHILE "->" IN curs REPEAT
        curs:=curs LEFTCUT "->";
        compte:=SUCC compte
      REPETITION
    TAKE compte DONE;

  OBJECT regle
   (mot_non_terminal VALUE gauche; mot VALUE droite);
  Boolean FUNCTION prior
   (regle VALUE r_1,r_2)
  DECLARE
    integer VALUE diff_1=LENGTH r_1.droite-LENGTH r_1.gauche,
                  diff_2=LENGTH r_2.droite-LENGTH r_2.gauche
  DO(*prior*)TAKE
    UNLESS diff_1=diff_2 THEN
      diff_1>diff_2
    DEFAULT LENGTH r_2.gauche<LENGTH r_1.gauche DONE
  DONE(*prior*);

  regle ROW rangee_de_regles VALUE regle_table=
    DECLARE
      string VARIABLE curs:=regles;
      rangee_de_regles VALUE rt=
        THROUGH
          rangee_de_regles(1 TO nombre_regles)
          := regle((curs:=curs LEFTCUT " ") LEFTOF "->",
                   (curs:=curs LEFTCUT "->") LEFTOF " ")
        REPETITION;

      PROCEDURE tri_sec(integer VALUE inf,sup) DECLARE
        integer VARIABLE bas:=inf, haut:=SUCC sup;
        regle VALUE r=rt[bas]
      DO(*tri_sec*)
        CYCLE zig_zag REPEAT
          WHILE prior(r,rt[(haut:=PRED haut)]) REPETITION;
        IF bas=haut EXIT zig_zag DONE;
          rt[bas]:=:rt[haut];
          WHILE prior(rt[(bas:=SUCC bas)],r) REPETITION;
        IF bas=haut EXIT zig_zag DONE;
```

```
ascendant                                          Vax Newton Compiler 0.2c9
                                          Page    3
                    Source listing

                   rt[bas]:=:rt[haut]
                 REPETITION;
                 IF inf<(bas:=PRED bas) THEN tri_sec(inf,bas) DONE;
                 IF (haut:=SUCC haut)<sup THEN tri_sec(haut,sup) DONE
               DONE(*tri_sec*)
            DO tri_sec(1,nombre_regles) TAKE rt DONE
        DO(*decoupage_regles*)DONE;
        /* /*EJECT*/ */
```

Ce deuxième critère heuristique implique que l'on estime qu'une substitution possible d'un mot partiel long a plus de chances d'être définitive.

Dans la partie exécutable du programme *ascendant*, le fichier donné est lu; les mots terminaux en sont extraits et confiés à l'algorithme d'analyse syntaxique incorporé dans la procédure *analyse*. La partie exécutable de cette dernière comporte essentiellement un cycle de structure très typique pour le contrôle du retour arrière. On remarque que la traînée est déclarée sous la forme d'une variable de manière à rendre cette pile extensible. Dans cette pile sont incorporés, après chaque réduction, une coroutine du type *parents* chargée d'engendrer chacun des mots susceptibles d'être obtenus au moyen d'une seule réduction à partir du mot qui lui est fourni comme paramètre lors de sa création.

```
ascendant                                  Vax Newton Compiler 0.2c9
                                        Page     4
                    Source listing

   PROCESS parents  VALUE mes_parents
     ATTRIBUTE sujet,courant,prochain
     (mot VALUE sujet)
   DECLARE(*parents*)
     mot VARIABLE cour VALUE courant;
     mot EXPRESSION prochain=
       (ACTIVATE mes_parents NOW; courant);

     PROCEDURE applique(regle VALUE r) DECLARE
       mot VARIABLE debut:="",reste:=sujet
     DO
       CONNECT r THEN
         UNLESS LENGTH droite=0 THEN
           WHILE droite IN reste REPEAT
           RETURN
             cour:=
                (debut:=debut+reste LEFTOF droite)
               +gauche
               +(reste:=reste LEFTCUT droite);
             debut:=debut+droite CHAR 1;
             reste:=droite@2+reste
           REPETITION
         DEFAULT
           FOR integer VALUE pos FROM 0 TO LENGTH sujet REPEAT
           RETURN
             cour:=sujet..pos+gauche+sujet@(SUCC pos)
           REPETITION
         DONE
       DONE(*CONNECT r*)
     DONE(*applique*)
   DO(*parents*)
     THROUGH
       regle_table VALUE regle_courante
     REPEAT applique(regle_courante) REPETITION
   DONE(*parents*);

   PROCEDURE analyse(mot_terminal VALUE sujet) DECLARE
     string VARIABLE ancetres:=" "+sujet+" ";
     parents STACK filiaire VARIABLE trainee:=
       (filiaire(nombre_regles) PUSH parents(sujet));
     CONSTANT increment=1.25;
     PROCEDURE restructure_trainee DO
       THROUGH
         (filiaire(increment*CAPACITY trainee)=:trainee)
         VALUE ancetre
       REPEAT trainee PUSH ancetre REPETITION
     DONE(*restructure_trainee*);

     PROCEDURE imprime_longue_chaine
       (string VARIABLE sujet)
     DECLARE
       CONSTANT max_ligne=65;
```

```
        string VARIABLE debut
      DO(*imprime_longue_chaine*)
        WHILE LENGTH sujet>max_ligne REPEAT
          print(_____,(debut:=sujet..max_ligne RIGHTCUT " "),line);
          sujet:=" "-sujet LEFTCUT debut
        REPETITION;
        print(_____,sujet,line,__"<<<FIN>>>",line)
      DONE(*imprime_longue_chaine*);
      parents VARIABLE generation
    DO(*analyse*)
      print("*****Analyse du mot--->",sujet,"<---*****",line);
      CYCLE cree_ancetres REPEAT
      IF STATE trainee TOP=terminated THEN
        generation POP trainee;
        IF EMPTY trainee THEN
          print(__,"###--->",sujet,
              "<---ne fait pas partie de la grammaire###",line);
          print(__,"***Liste des ancetres trouves***",line);
          imprime_longue_chaine(" "-ancetres)
        EXIT cree_ancetres DONE;
      REPEAT cree_ancetres DONE;
      IF
        " "+(trainee TOP).prochain+" " IN ancetres
      REPEAT cree_ancetres DONE;
      IF (trainee TOP).courant=axiome THEN
        print(__,"***--->",sujet,
             "<---est engendre par la grammaire***",line);
        ancetres:="";
        UNTIL EMPTY trainee REPEAT
          generation POP trainee;
          ancetres:=ancetres+generation.courant+" "
        REPETITION;
        print(__,"***Derivation trouvee***",line);
        imprime_longue_chaine(ancetres+sujet)
      EXIT cree_ancetres DONE;
        ancetres:=ancetres+(trainee TOP).courant+" ";
        IF FULL trainee THEN restructure_trainee DONE;
        trainee PUSH parents((trainee TOP).courant)
      REPETITION(*CYCLE cree_ancetres*)
    DONE(*analyse*);
  /* /*EJECT*/ */
```

```
ascendant                                        Vax Newton Compiler 0.2c9
                                         Page    6
                      Source listing

     string VARIABLE ligne
   DO(*ascendant*)
     UNTIL end_file REPEAT
       read(ligne);
       WHILE terminal<>ligne REPEAT
         analyse(terminal SPAN (ligne:=ligne ATLEFT terminal));
         ligne:=terminal-ligne
       REPETITION
     REPETITION;
     print("<<<<<FIN DES ANALYSES>>>>>")
   DONE(*ascendant*)

     ****  No messages were issued  ****
```

Dans la procédure *analyse*, on remarque aussi le rôle de la variable *ancêtres*. Cette chaîne contient tous les mots que l'on a pu obtenir, au moyen de réductions, à partir du mot *sujet* à analyser: ces mots y sont séparés par des espaces blancs. Cette variable *ancêtres* est notamment utilisée pour déceler des cycles éventuels; un tel cycle pourrait intervenir si la grammaire est ambiguë. Si l'on est sûr que la grammaire à analyser n'est pas ambiguë, on pourrait se passer de cette variable, ce qui permettra d'accélérer le traitement.
Lorsqu'il est prouvé que le mot à analyser est engendré par la grammaire, la dérivation trouvée est immédiatement obtenue en dépilant la traînée et en extrayant, de chacun de ses membres, la chaîne *courant*. Les mots successifs de la dérivation sont alors produits de l'axiome au mot à analyser.

Résultats:

```
*****Analyse du mot--->x<---*****
  ***--->x<---est engendre par la grammaire***
  ***Derivation trouvee***
    E T F B V x
  <<<FIN>>>
*****Analyse du mot--->f(g(-x))<---*****
  ***--->f(g(-x))<---est engendre par la grammaire***
  ***Derivation trouvee***
    E T F B D(E) D(T) D(F) D(B) D(D(E)) D(D(AT)) D(D(AF)) D(D(AB))
    D(D(AV)) D(D(-V)) D(D(-x)) D(g(-x)) f(g(-x))
  <<<FIN>>>
*****Analyse du mot--->(x-y+z)<---*****
  ***--->(x-y+z)<---est engendre par la grammaire***
  ***Derivation trouvee***
    E T F B (E) (EAT) (EAF) (EAB) (EATAB) (EAFAB) (EABAB) (TABAB)
    (FABAB) (BABAB) (BABAV) (BAVAV) (VAVAV) (V-VAV) (V-V+V) (V-V+z)
    (V-y+z) (x-y+z)
  <<<FIN>>>
*****Analyse du mot--->(-z^y^x)<---*****
  ***--->(-z^y^x)<---est engendre par la grammaire***
  ***Derivation trouvee***
    E T F B (E) (AT) (AF) (AB^F) (AB^B^F) (AB^B^B) (AB^B^V) (AB^V^V)
    (AV^V^V) (-V^V^V) (-z^V^V) (-z^y^V) (-z^y^x)
  <<<FIN>>>
*****Analyse du mot--->((x-y)/(x+y))<---*****
  ***--->((x-y)/(x+y))<---est engendre par la grammaire***
  ***Derivation trouvee***
```

```
    E T F B (E) (T) (TMF) (T/F) (T/B) (T/(E)) (T/(EAT)) (T/(EAF))
    (T/(EAB)) (T/(TAB)) (T/(FAB)) (T/(BAB)) (F/(BAB)) (B/(BAB))
    ((E)/(BAB)) ((EAT)/(BAB)) ((EAF)/(BAB)) ((EAB)/(BAB))
    ((TAB)/(BAB)) ((FAB)/(BAB)) ((BAB)/(BAB)) ((BAB)/(BAV))
    ((BAB)/(VAV)) ((BAV)/(VAV)) ((VAV)/(VAV)) ((V-V)/(VAV))
    ((V-V)/(V+V)) ((V-V)/(V+y)) ((V-y)/(V+y)) ((V-y)/(x+y))
    ((x-y)/(x+y))
  <<<FIN>>>
*****Analyse du mot--->(x-y)*z<---*****
  ###--->(x-y)*z<---ne fait pas partie de la grammaire###
  ***Liste des ancetres trouves***
    (x-y)*z (V-y)*z (V-V)*z (V-V)*V (VAV)*V (BAV)*V (BAB)*V (BAB)*B
    (FAB)*B (TAB)*B (EAB)*B (EAF)*B (EAT)*B (E)*B B*B F*B T*B E*B
    E*F E*T E*E EME EMT EMF EMB T*F T*T T*E TME TMT TMF TMB F*F F*T
    F*E FME FMT FMF FMB B*F B*T B*E BME BMT BMF BMB (E)*F (E)*T
    (E)*E (E)ME (E)MT (E)MF (E)MB (EAE)*B (EAE)*F (EAE)*T (EAE)*E
    (EAE)ME (EAE)MT (EAE)MF (EAE)MB (EAT)*F (EAT)*T (EAT)*E (EAT)ME
    (EAT)MT (EAT)MF (EAT)MB (EAF)*F (EAF)*T (EAF)*E (EAF)ME (EAF)MT
    (EAF)MF (EAF)MB (EAB)*F (EAB)*T (EAB)*E (EAB)ME (EAB)MT (EAB)MF
    (EAB)MB (TAF)*B (TAT)*B (TAE)*B (TAE)*F (TAE)*T (TAE)*E (TAE)ME
    (TAE)MT (TAE)MF (TAE)MB (TAT)*F (TAT)*T (TAT)*E (TAT)ME (TAT)MT
    (TAT)MF (TAT)MB (TAF)*F (TAF)*T (TAF)*E (TAF)ME (TAF)MT (TAF)MF
    (TAF)MB (TAB)*F (TAB)*T (TAB)*E (TAB)ME (TAB)MT (TAB)MF (TAB)MB
    (FAF)*B (FAT)*B (FAE)*B (FAE)*F (FAE)*T (FAE)*E (FAE)ME (FAE)MT
    (FAE)MF (FAE)MB (FAT)*F (FAT)*T (FAT)*E (FAT)ME (FAT)MT (FAT)MF
    (FAT)MB (FAF)*F (FAF)*T (FAF)*E (FAF)ME (FAF)MT (FAF)MF (FAF)MB
    (FAB)*F (FAB)*T (FAB)*E (FAB)ME (FAB)MT (FAB)MF (FAB)MB (BAF)*B
    (BAT)*B (BAE)*B (BAE)*F (BAE)*T (BAE)*E (BAE)ME (BAE)MT (BAE)MF
    (BAE)MB (BAT)*F (BAT)*T (BAT)*E (BAT)ME (BAT)MT (BAT)MF (BAT)MB
    (BAF)*F (BAF)*T (BAF)*E (BAF)ME (BAF)MT (BAF)MF (BAF)MB (BAB)*F
    (BAB)*T (BAB)*E (BAB)ME (BAB)MT (BAB)MF (BAB)MB (FAB)*V (TAB)*V
    (EAB)*V (EAF)*V (EAT)*V (E)*V B*V F*V T*V E*V EMV TMV FMV BMV
    (E)MV (EAE)*V (EAE)MV (EAT)MV (EAF)MV (EAB)MV (TAF)*V (TAT)*V
    (TAE)*V (TAE)MV (TAT)MV (TAF)MV (TAB)MV (FAF)*V (FAT)*V (FAE)*V
    (FAE)MV (FAT)MV (FAF)MV (FAB)MV (BAF)*V (BAT)*V (BAE)*V (BAE)MV
    (BAT)MV (BAF)MV (BAB)MV (BAV)*B (FAV)*B (TAV)*B (EAV)*B (EAV)*F
    (EAV)*T (EAV)*E (EAV)ME (EAV)MT (EAV)MF (EAV)MB (TAV)*F (TAV)*T
    (TAV)*E (TAV)ME (TAV)MT (TAV)MF (TAV)MB (FAV)*F (FAV)*T (FAV)*E
    (FAV)ME (FAV)MT (FAV)MF (FAV)MB (BAV)*F (BAV)*T (BAV)*E (BAV)ME
    (BAV)MT (BAV)MF (BAV)MB (FAV)*V (TAV)*V (EAV)*V (EAV)MV (TAV)MV
    (FAV)MV (BAV)MV (VAB)*V (VAB)*B (VAF)*B (VAT)*B (VAE)*B (VAE)*F
    (VAE)*T (VAE)*E (VAE)ME (VAE)MT (VAE)MF (VAE)MB (VAT)*F (VAT)*T
    (VAT)*E (VAT)ME (VAT)MT (VAT)MF (VAT)MB (VAF)*F (VAF)*T (VAF)*E
    (VAF)ME (VAF)MT (VAF)MF (VAF)MB (VAB)*F (VAB)*T (VAB)*E (VAB)ME
    (VAB)MT (VAB)MF (VAB)MB (VAF)*V (VAT)*V (VAE)*V (VAE)MV (VAT)MV
    (VAF)MV (VAB)MV (VAV)*B (VAV)*F (VAV)*T (VAV)*E (VAV)ME (VAV)MT
    (VAV)MF (VAV)MB (VAV)MV (B-V)*V (B-B)*V (B-B)*B (F-B)*B (T-B)*B
    (E-B)*B (E-F)*B (E-T)*B (E-E)*B (E-E)*F (E-E)*T (E-E)*E (E-E)ME
    (E-E)MT (E-E)MF (E-E)MB (E-T)*F (E-T)*T (E-T)*E (E-T)ME (E-T)MT
    (E-T)MF (E-T)MB (E-F)*F (E-F)*T (E-F)*E (E-F)ME (E-F)MT (E-F)MF
    (E-F)MB (E-B)*F (E-B)*T (E-B)*E (E-B)ME (E-B)MT (E-B)MF (E-B)MB
    (T-F)*B (T-T)*B (T-E)*B (T-E)*F (T-E)*T (T-E)*E (T-E)ME (T-E)MT
    (T-E)MF (T-E)MB (T-T)*F (T-T)*T (T-T)*E (T-T)ME (T-T)MT (T-T)MF
    (T-T)MB (T-F)*F (T-F)*T (T-F)*E (T-F)ME (T-F)MT (T-F)MF (T-F)MB
    (T-B)*F (T-B)*T (T-B)*E (T-B)ME (T-B)MT (T-B)MF (T-B)MB (F-F)*B
    (F-T)*B (F-E)*B (F-E)*F (F-E)*T (F-E)*E (F-E)ME (F-E)MT (F-E)MF
    (F-E)MB (F-T)*F (F-T)*T (F-T)*E (F-T)ME (F-T)MT (F-T)MF (F-T)MB
    (F-F)*F (F-F)*T (F-F)*E (F-F)ME (F-F)MT (F-F)MF (F-F)MB (F-B)*F
    (F-B)*T (F-B)*E (F-B)ME (F-B)MT (F-B)MF (F-B)MB (B-F)*B (B-T)*B
    (B-E)*B (B-E)*F (B-E)*T (B-E)*E (B-E)ME (B-E)MT (B-E)MF (B-E)MB
    (B-T)*F (B-T)*T (B-T)*E (B-T)ME (B-T)MT (B-T)MF (B-T)MB (B-F)*F
    (B-F)*T (B-F)*E (B-F)ME (B-F)MT (B-F)MF (B-F)MB (B-B)*F (B-B)*T
    (B-B)*E (B-B)ME (B-B)MT (B-B)MF (B-B)MB (F-B)*V (T-B)*V (E-B)*V
    (E-F)*V (E-T)*V (E-E)*V (E-E)MV (E-T)MV (E-F)MV (E-B)MV (T-F)*V
```

```
        (T-T)*V  (T-E)*V  (T-E)MV  (T-T)MV  (T-F)MV  (T-B)MV  (F-F)*V  (F-T)*V
        (F-E)*V  (F-E)MV  (F-T)MV  (F-F)MV  (F-B)MV  (B-F)*V  (B-T)*V  (B-E)*V
        (B-E)MV  (B-T)MV  (B-F)MV  (B-B)MV  (B-V)*B  (F-V)*B  (T-V)*B  (E-V)*B
        (E-V)*F  (E-V)*T  (E-V)*E  (E-V)ME  (E-V)MT  (E-V)MF  (E-V)MB  (T-V)*F
        (T-V)*T  (T-V)*E  (T-V)ME  (T-V)MT  (T-V)MF  (T-V)MB  (F-V)*F  (F-V)*T
        (F-V)*E  (F-V)ME  (F-V)MT  (F-V)MF  (F-V)MB  (B-V)*F  (B-V)*T  (B-V)*E
        (B-V)ME  (B-V)MT  (B-V)MF  (B-V)MB  (F-V)*V  (T-V)*V  (E-V)*V  (E-V)MV
        (T-V)MV  (F-V)MV  (B-V)MV  (V-B)*V  (V-B)*B  (V-F)*B  (V-T)*B  (V-E)*B
        (V-E)*F  (V-E)*T  (V-E)*E  (V-E)ME  (V-E)MT  (V-E)MF  (V-E)MB  (V-T)*F
        (V-T)*T  (V-T)*E  (V-T)ME  (V-T)MT  (V-T)MF  (V-T)MB  (V-F)*F  (V-F)*T
        (V-F)*E  (V-F)ME  (V-F)MT  (V-F)MF  (V-F)MB  (V-B)*F  (V-B)*T  (V-B)*E
        (V-B)ME  (V-B)MT  (V-B)MF  (V-B)MB  (V-F)*V  (V-T)*V  (V-E)*V  (V-E)MV
        (V-T)MV  (V-F)MV  (V-B)MV  (V-V)*B  (V-V)*F  (V-V)*T  (V-V)*E  (V-V)ME
        (V-V)MT  (V-V)MF  (V-V)MB  (V-V)MV  (VAV)*z  (BAV)*z  (BAB)*z  (FAB)*z
        (TAB)*z  (EAB)*z  (EAF)*z  (EAT)*z  (E)*z B*z F*z  T*z  E*z  EMz  TMz
        FMz BMz  (E)Mz  (EAE)*z  (EAE)Mz  (EAT)Mz  (EAF)Mz  (EAB)Mz  (TAF)*z
        (TAT)*z  (TAE)*z  (TAE)Mz  (TAT)Mz  (TAF)Mz  (TAB)Mz  (FAF)*z  (FAT)*z
        (FAE)*z  (FAE)Mz  (FAT)Mz  (FAF)Mz  (FAB)Mz  (BAF)*z  (BAT)*z  (BAE)*z
        (BAE)Mz  (BAT)Mz  (BAF)Mz  (BAB)Mz  (FAV)*z  (TAV)*z  (EAV)*z  (EAV)Mz
        (TAV)Mz  (FAV)Mz  (BAV)Mz  (VAB)*z  (VAF)*z  (VAT)*z  (VAE)*z  (VAE)Mz
        (VAT)Mz  (VAF)Mz  (VAB)Mz  (VAV)Mz  (B-V)*z  (B-B)*z  (F-B)*z  (T-B)*z
        (E-B)*z  (E-F)*z  (E-T)*z  (E-E)*z  (E-E)Mz  (E-T)Mz  (E-F)Mz  (E-B)Mz
        (T-F)*z  (T-T)*z  (T-E)*z  (T-E)Mz  (T-T)Mz  (T-F)Mz  (T-B)Mz  (F-F)*z
        (F-T)*z  (F-E)*z  (F-E)Mz  (F-T)Mz  (F-F)Mz  (F-B)Mz  (B-F)*z  (B-T)*z
        (B-E)*z  (B-E)Mz  (B-T)Mz  (B-F)Mz  (B-B)Mz  (F-V)*z  (T-V)*z  (E-V)*z
        (E-V)Mz  (T-V)Mz  (F-V)Mz  (B-V)Mz  (V-B)*z  (V-F)*z  (V-T)*z  (V-E)*z
        (V-E)Mz  (V-T)Mz  (V-F)Mz  (V-B)Mz  (V-V)Mz  (V-y)*V  (VAy)*V  (BAy)*V
        (BAy)*B  (FAy)*B  (TAy)*B  (EAy)*B  (EAy)*F  (EAy)*T  (EAy)*E  (EAy)ME
        (EAy)MT  (EAy)MF  (EAy)MB  (TAy)*F  (TAy)*T  (TAy)*E  (TAy)ME  (TAy)MT
        (TAy)MF  (TAy)MB  (FAy)*F  (FAy)*T  (FAy)*E  (FAy)ME  (FAy)MT  (FAy)MF
        (FAy)MB  (BAy)*F  (BAy)*T  (BAy)*E  (BAy)ME  (BAy)MT  (BAy)MF  (BAy)MB
        (FAy)*V  (TAy)*V  (EAy)*V  (EAy)MV  (TAy)MV  (FAy)MV  (BAy)MV  (VAy)*B
        (VAy)*F  (VAy)*T  (VAy)*E  (VAy)ME  (VAy)MT  (VAy)MF  (VAy)MB  (VAy)MV
        (B-y)*V  (B-y)*B  (F-y)*B  (T-y)*B  (E-y)*B  (E-y)*F  (E-y)*T  (E-y)*E
        (E-y)ME  (E-y)MT  (E-y)MF  (E-y)MB  (T-y)*F  (T-y)*T  (T-y)*E  (T-y)ME
        (T-y)MT  (T-y)MF  (T-y)MB  (F-y)*F  (F-y)*T  (F-y)*E  (F-y)ME  (F-y)MT
        (F-y)MF  (F-y)MB  (B-y)*F  (B-y)*T  (B-y)*E  (B-y)ME  (B-y)MT  (B-y)MF
        (B-y)MB  (F-y)*V  (T-y)*V  (E-y)*V  (E-y)MV  (T-y)MV  (F-y)MV  (B-y)MV
        (V-y)*B  (V-y)*F  (V-y)*T  (V-y)*E  (V-y)ME  (V-y)MT  (V-y)MF  (V-y)MB
        (V-y)MV  (VAy)*z  (BAy)*z  (FAy)*z  (TAy)*z  (EAy)*z  (EAy)Mz  (TAy)Mz
        (FAy)Mz  (BAy)Mz  (VAy)Mz  (B-y)*z  (F-y)*z  (T-y)*z  (E-y)*z  (E-y)Mz
        (T-y)Mz  (F-y)Mz  (B-y)Mz  (V-y)Mz  (x-V)*z  (x-V)*V  (xAV)*V  (xAB)*V
        (xAB)*B  (xAF)*B  (xAT)*B  (xAE)*B  (xAE)*F  (xAE)*T  (xAE)*E  (xAE)ME
        (xAE)MT  (xAE)MF  (xAE)MB  (xAT)*F  (xAT)*T  (xAT)*E  (xAT)ME  (xAT)MT
        (xAT)MF  (xAT)MB  (xAF)*F  (xAF)*T  (xAF)*E  (xAF)ME  (xAF)MT  (xAF)MF
        (xAF)MB  (xAB)*F  (xAB)*T  (xAB)*E  (xAB)ME  (xAB)MT  (xAB)MF  (xAB)MB
        (xAF)*V  (xAT)*V  (xAE)*V  (xAE)MV  (xAT)MV  (xAF)MV  (xAB)MV  (xAV)*B
        (xAV)*F  (xAV)*T  (xAV)*E  (xAV)ME  (xAV)MT  (xAV)MF  (xAV)MB  (xAV)MV
        (x-B)*V  (x-B)*B  (x-F)*B  (x-T)*B  (x-E)*B  (x-E)*F  (x-E)*T  (x-E)*E
        (x-E)ME  (x-E)MT  (x-E)MF  (x-E)MB  (x-T)*F  (x-T)*T  (x-T)*E  (x-T)ME
        (x-T)MT  (x-T)MF  (x-T)MB  (x-F)*F  (x-F)*T  (x-F)*E  (x-F)ME  (x-F)MT
        (x-F)MF  (x-F)MB  (x-B)*F  (x-B)*T  (x-B)*E  (x-B)ME  (x-B)MT  (x-B)MF
        (x-B)MB  (x-F)*V  (x-T)*V  (x-E)*V  (x-E)MV  (x-T)MV  (x-F)MV  (x-B)MV
        (x-V)*B  (x-V)*F  (x-V)*T  (x-V)*E  (x-V)ME  (x-V)MT  (x-V)MF  (x-V)MB
        (x-V)MV  (xAV)*z  (xAB)*z  (xAF)*z  (xAT)*z  (xAE)*z  (xAE)Mz  (xAT)Mz
        (xAF)Mz  (xAB)Mz  (xAV)Mz  (x-B)*z  (x-F)*z  (x-T)*z  (x-E)*z  (x-E)Mz
        (x-T)Mz  (x-F)Mz  (x-B)Mz  (x-V)Mz  (x-y)*V  (xAy)*V  (xAy)*B  (xAy)*F
        (xAy)*T  (xAy)*E  (xAy)ME  (xAy)MT  (xAy)MF  (xAy)MB  (xAy)MV  (x-y)*B
        (x-y)*F  (x-y)*T  (x-y)*E  (x-y)ME  (x-y)MT  (x-y)MF  (x-y)MB  (x-y)MV
        (xAy)*z  (xAy)Mz  (x-y)Mz
   <<<FIN>>>
*****Analyse du mot--->h(x^y-z)<---*****
  ***--->h(x^y-z)<---est engendre par la grammaire***
  ***Derivation trouvee***
```

```
      E T F B D(E) D(EAT) D(EAF) D(EAB) D(TAB) D(FAB) D(B^FAB)
      D(B^BAB) D(B^BAV) D(B^VAV) D(V^VAV) D(V^V-V) D(V^V-z) D(V^y-z)
      D(x^y-z) h(x^y-z)
   <<<FIN>>>
*****Analyse du mot--->(x*y+z^h(z))<---*****
   ***--->(x*y+z^h(z))<---est engendre par la grammaire***
   ***Derivation trouvee***
      E T F B (E) (EAT) (TAT) (TMFAT) (T*FAT) (T*FAF) (T*FAB^F)
      (T*FAB^B) (T*FAB^D(E)) (T*FAB^D(T)) (T*FAB^D(F)) (T*FAB^D(B))
      (T*BAB^D(B)) (F*BAB^D(B)) (B*BAB^D(B)) (B*BAB^D(V)) (B*BAV^D(V))
      (B*VAV^D(V)) (V*VAV^D(V)) (V*V+V^D(V)) (V*V+V^D(z)) (V*V+z^D(z))
      (V*y+z^D(z)) (x*y+z^D(z)) (x*y+z^h(z))
   <<<FIN>>>
*****Analyse du mot--->(x^y/h(z)/y)<---*****
   ***--->(x^y/h(z)/y)<---est engendre par la grammaire***
   ***Derivation trouvee***
      E T F B (E) (T) (TMF) (T/F) (TMF/F) (FMF/F) (B^FMF/F) (B^F/F/F)
      (B^F/F/B) (B^F/B/B) (B^F/D(E)/B) (B^F/D(T)/B) (B^F/D(F)/B)
      (B^F/D(B)/B) (B^B/D(B)/B) (B^B/D(B)/V) (B^B/D(V)/V) (B^V/D(V)/V)
      (V^V/D(V)/V) (V^V/D(z)/V) (V^V/D(z)/y) (V^y/D(z)/y) (x^y/D(z)/y)
      (x^y/h(z)/y)
   <<<FIN>>>
<<<<<FIN DES ANALYSES>>>>>
```

Lorsque le mot *sujet* n'est pas engendré par la grammaire, on remarque qu'il est produit, à la place d'une dérivation, l'ensemble des mots susceptibles d'être obtenus par réduction de ce mot.

Malgré l'heuristique choisie, cet algorithme est en général lent. Il ne permet d'analyser en un temps raisonnable que des mots très courts. Pour cette raison, les grammaires utilisées pour décrire la syntaxe d'un langage de programmation ont en général une forme permettant une analyse par des moyens plus simples. Il s'agit presque toujours de grammaires indépendantes du contexte (le membre gauche de chaque règle de production est un seul symbole non terminal); souvent, la grammaire a même une forme permettant d'adapter la méthode de descente récursive.
A priori, on pourrait chercher à modifier ce programme pour analyser une grammaire dont certaines règles de production ont un membre gauche plus long que le membre droit. Dans ce cas, certaines "réductions" pourront allonger le mot à analyser. Ce programme modifié produira les ancêtres d'un mot en largeur et non en profondeur, ce que l'on obtient en gérant la traînée en queue. On peut alors prouver les résultats suivants :

- Si le mot à analyser est engendré par la grammaire, il est garanti que l'algorithme le décèlera en un temps fini et produira une dérivation de ce mot à partir de l'axiome.

- Par contre, si le mot à analyser ne fait pas partie du langage engendré par la grammaire, il n'y a en général aucune garantie que le programme le décèlera en un temps fini. Pour certains mots, l'algorithme sera à même de déceler qu'ils ne sont pas engendrés par la grammaire. Pour d'autres par contre, le programme effectuera une suite infinie de réductions sans jamais être à même de conclure si le mot est engendré par la grammaire ou non.

C'est évidemment à cause de cette dernière remarque que le bât blesse. Si le programme consacre un temps très long à l'analyse d'un mot sans conclure, on ne sait pas si cela vient du fait que l'on est entré dans une suite de réductions ou si une conclusion interviendra éventuellement. On peut d'ailleurs montrer que dans le cas des grammaires les plus générales, il n'est tout simplement pas possible de réaliser un algorithme d'analyse syntaxique capable de conclure pour tout mot terminal, en un temps fini s'il est engendré ou non par la grammaire.

A titre comparatif, on va maintenant donner un programme d'analyse syntaxique de nature descendante (figure 35).

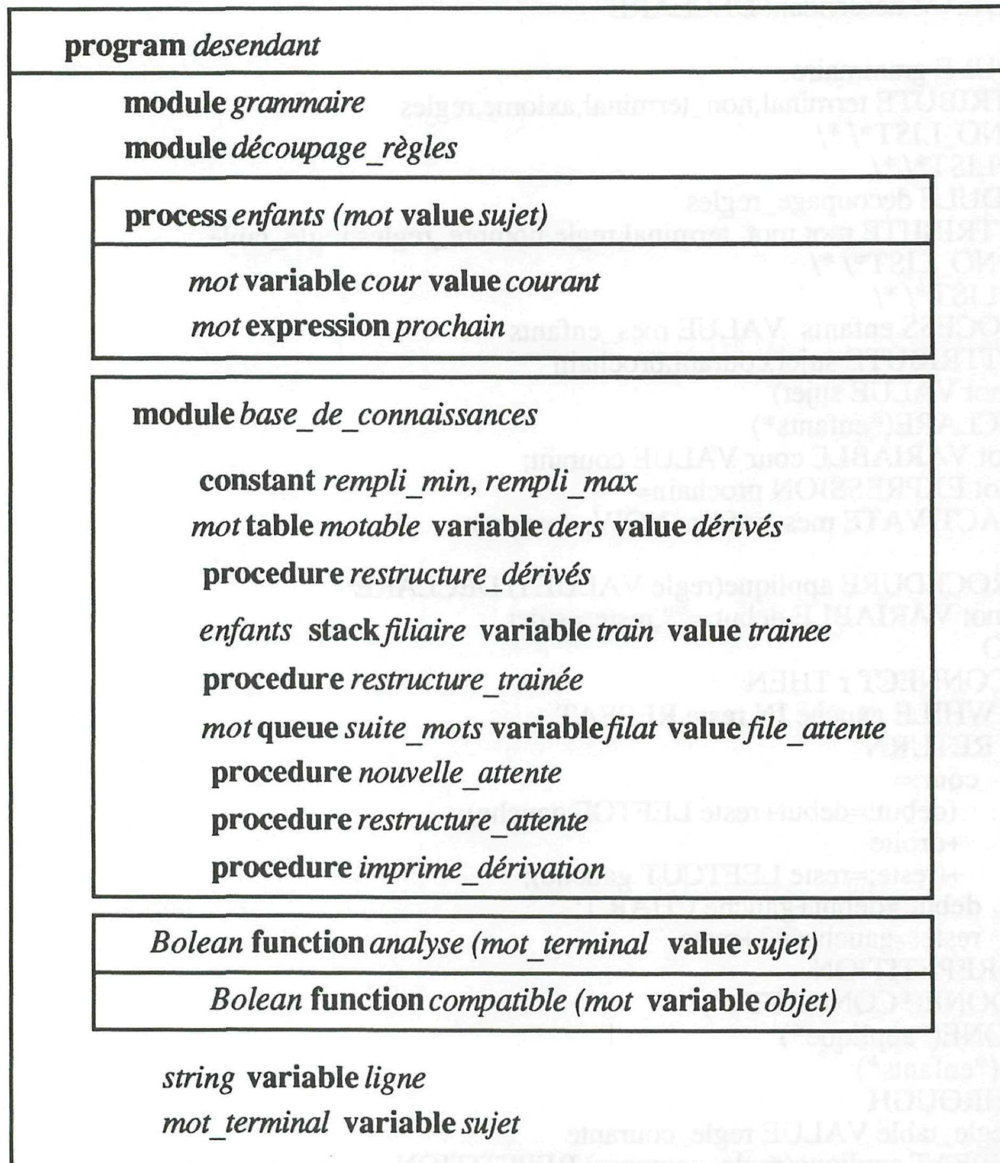

program desendant
module grammaire
module découpage_règles
process enfants (mot value sujet)
mot variable cour value courant
mot expression prochain
module base_de_connaissances
constant rempli_min, rempli_max
mot table motable variable ders value dérivés
procedure restructure_dérivés
enfants stack filiaire variable train value trainee
procedure restructure_trainée
mot queue suite_mots variable filat value file_attente
procedure nouvelle_attente
procedure restructure_attente
procedure imprime_dérivation
Bolean function analyse (mot_terminal value sujet)
Bolean function compatible (mot variable objet)
string variable ligne
mot_terminal variable sujet

Figure 35

descendant Vax Newton Compiler 0.2c14
Page 1
Source listing

```
/* /*OLDSOURCE=USER2:[RAPIN]DESCENDANT.NEW*/ */
PROGRAM descendant DECLARE

MODULE grammaire
 ATTRIBUTE terminal,non_terminal,axiome,regles
/* /*NO_LIST*/ */
/* /*LIST*/ */
MODULE decoupage_regles
 ATTRIBUTE mot,mot_terminal,regle,nombre_regles,regle_table
/* /*NO_LIST*/ */
/* /*LIST*/ */
 PROCESS enfants VALUE mes_enfants
  ATTRIBUTE sujet,courant,prochain
  (mot VALUE sujet)
 DECLARE(*enfants*)
  mot VARIABLE cour VALUE courant;
  mot EXPRESSION prochain=
  (ACTIVATE mes_enfants NOW; courant);

  PROCEDURE applique(regle VALUE r) DECLARE
   mot VARIABLE debut:="",reste:=sujet
  DO
   CONNECT r THEN
    WHILE gauche IN reste REPEAT
    RETURN
     cour:=
      (debut:=debut+reste LEFTOF gauche)
      +droite
      +(reste:=reste LEFTCUT gauche);
     debut:=debut+gauche CHAR 1;
     reste:=gauche@2+reste
    REPETITION
   DONE(*CONNECT r*)
  DONE(*applique*)
 DO(*enfants*)
  THROUGH
   regle_table VALUE regle_courante
  REPEAT applique(regle_courante) REPETITION
 DONE(*enfants*);
/* /*EJECT*/ */
```

Ce programme *descendant* analyse la même grammaire que le programme *ascendant*; les modules *grammaire* et *découpage_règles* sont identiques que dans ce dernier. Le type processus *enfants* est à peu de chose près le symétrique de *parents* dans le premier programme. Un membre gauche d'une règle de production ne pouvant être vide, il n'a cependant été pris aucune précaution à ce sujet.

descendant Vax Newton Compiler 0.2c14
Page 2
Source listing

```
MODULE base_de_connaissances
 ATTRIBUTE
   derives,restructure_derives,rempli_min,rempli_max,
   imprime_derivation,
   trainee,restructure_trainee,
   suite_mots,file_attente,restructure_attente,nouvelle_attente
DECLARE(*base_de_connaissances*)
 CONSTANT taille_init=20,increment=1.25,rempli_min=.5,rempli_max=.8;

 mot TABLE motable VARIABLE ders
   VALUE derives:=motable(taille_init);
 PROCEDURE restructure_derives DO
   THROUGH
    (motable(CARD derives/rempli_min)=:ders)
    INDEX mot VALUE pere
   REPEAT derives[mot]:=pere REPETITION
 DONE(*restructure_derives*);

 enfants STACK filiaire VARIABLE train
   VALUE trainee:=filiaire(taille_init);
 PROCEDURE restructure_trainee DO
   THROUGH
    (filiaire(increment*CAPACITY trainee)=:train)
    VALUE ancetre
   REPEAT trainee PUSH ancetre REPETITION
 DONE(*restructure_trainee*);

 mot QUEUE suite_mots VARIABLE filat
   VALUE file_attente:=(suite_mots(taille_init) APPEND axiome);
 PROCEDURE nouvelle_attente DO
   filat:=suite_mots(taille_init)
 DONE(*nouvelle_attente*);
 PROCEDURE restructure_attente DO
   THROUGH
    (suite_mots(increment*CAPACITY file_attente)=:filat)
    VALUE suspendu
   REPEAT file_attente APPEND suspendu REPETITION
 DONE(*restructure_attente*);

 PROCEDURE imprime_longue_chaine
   (string VARIABLE sujet)
 DECLARE
   CONSTANT max_ligne=65;
   string VARIABLE debut
 DO(*imprime_longue_chaine*)
   WHILE LENGTH sujet>max_ligne REPEAT
     print(_____,(debut:=sujet..max_ligne RIGHTCUT " "),line);
     sujet:=" "-sujet LEFTCUT debut
   REPETITION;
    print(_____,sujet,line,__"<<<FIN>>>",line)
 DONE(*imprime_longue_chaine*);
```

```
descendant                    Vax Newton Compiler 0.2c14
                       Page    3
         Source listing

PROCEDURE imprime_derivation
  (mot VARIABLE sujet)
DECLARE string VARIABLE derivation:="" DO
  print(line,#***Derivation du mot "#,sujet,#"***#,line);
  UNTIL
    derivation:=sujet+" "+derivation
  TAKE sujet=axiome REPEAT
    sujet:=derives[sujet]
  REPETITION;
  imprime_longue_chaine(derivation)
DONE(*imprime_derivation*)
BEGIN(*base_de_connaissances*)
  derives[axiome]:=""
INNER
  print(page,"*****Derives trouves*****",line);
  DECLARE
    string VARIABLE mots_terminaux,mots_non_terminaux:=""
  DO
    THROUGH derives INDEX mot REPEAT
      IF mot IN terminal THEN
        mots_terminaux:=mots_terminaux+mot+" "
      DEFAULT
        mots_non_terminaux:=mots_non_terminaux+mot+" "
      DONE
    REPETITION;
    print(line,"***Mots terminaux engendres par la grammaire***",line);
    imprime_longue_chaine(mots_terminaux);
    print(line,"***Mots derives auxiliaires***",line);
    imprime_longue_chaine(mots_non_terminaux)
  DONE(*DECLARE ...*);
  print(line,"<<<<<FIN>>>>>")
END(*base_de_connaissances*);
/* /*EJECT*/ */
```

Le procédé d'analyse descendante utilisé va produire des dérivations qui ne seront pas toutes utiles pour l'analyse du mot considéré mais qui pourront se révéler utiles plus tard, pour l'analyse d'autres mots. Pour cela, il y a intérêt de conserver d'une analyse à l'autre, les informations (en particulier les dérivations) obtenues; c'est-là le rôle du module *base_de_connaissances*. Ce module comporte trois structures de données :

- La table extensible de mots *dérivés*. Cette table contient une entrée pour chaque mot (terminal ou non) que l'on a pu dériver de l'axiome; étant donné un mot *der*, l'entrée *dérivés [der]* a pour valeur le mot au moyen duquel ce mot *der* a été dérivé par l'application d'une règle unique. Cette information permettra de retrouver la dérivation des mots terminaux analysés; c'est la procédure *imprime_dérivation* qui s'en charge.

- La pile extensible *traînée*; pour la raison indiquée précédemment, la traînée est conservée d'une analyse à la suivante.

- La queue extensible de mots *file_attente*; sont insérés dans cette queue les mots auxquels on a arrêté (provisoirement) la production de dérivés parce qu'il a été reconnu que les mots terminaux que l'on a analysé à ce stade ne pouvaient dériver d'eux. Initialement, cette queue contient l'axiome de la grammaire.

On remarque qu'à la fin de l'exécution du programme, l'épilogue de ce module fait imprimer l'ensemble des mots terminaux et non terminaux que l'on a pu dériver de l'axiome.

```
descendant                    Vax Newton Compiler 0.2c14
                     Page  4
            Source listing

 981   Boolean FUNCTION analyse(mot_terminal VALUE sujet) DECLARE
 990
 990      Boolean FUNCTION compatible
 993        (mot VARIABLE objet)
 998      (*Faux si on peut exclure que le mot  sujet  soit derive
 998        du mot objet.
 998      *)
 998      DECLARE
 999        mot VARIABLE reste:=sujet;
1005        character VARIABLE symbole
1008       DO(*compatible*)TAKE
1010        CYCLE examen REPEAT
1013
1013          UNLESS
1014            LENGTH objet<=LENGTH reste
1019          EXIT examen TAKE FALSE DONE;
1025
1025          IF
1026            objet ATLEFT terminal=""
1031          EXIT examen TAKE TRUE DONE;
1037
1037          UNLESS
1038              (symbole:=objet LEFTOCC terminal)
1045            IN (reste:=reste@objet LEFTPOS symbole)
1055          EXIT examen TAKE FALSE DONE;
1061
1061            objet:=objet LEFTCUT symbole;
1067            reste:=reste LEFTCUT symbole
1072          REPETITION
1073       DONE(*compatible*);
1075
1075       enfants  VARIABLE avorton;
1079       suite_mots VARIABLE candidats;
1083       mot VARIABLE derive
1086    DO(*analyse*)TAKE
1088       IF derives ENTRY sujet THEN
1093         TRUE
1094       DEFAULT
1095         (*Le mot n'a pas encore ete trouve*)
1095         candidats:=file_attente;
1099         nouvelle_attente
1100       TAKE
1101         CYCLE cree_descendants REPEAT
1104           UNLESS EMPTY candidats THEN
1108             derive FROM candidats;
1112             UNLESS compatible(derive) THEN
1118               IF FULL file_attente THEN
1122                 restructure_attente
1123               DONE;
1125               file_attente APPEND derive
1128             REPEAT cree_descendants DONE;
1132
```

```
descendant                    Vax Newton Compiler 0.2c14
                         Page    5
          Source listing

              IF FULL trainee THEN
               restructure_trainee
              DONE;
              trainee PUSH enfants(derive) ELSE
             IF
              EMPTY trainee
             EXIT cree_descendants TAKE FALSE DONE;
             CYCLE traite_trainee REPEAT
              (*~EMPTY trainee*)
              IF STATE trainee TOP=terminated THEN
               avorton POP trainee
              REPEAT cree_descendants DONE;
              UNLESS
               derives ENTRY (derive:=(trainee TOP).prochain)
              THEN
               derives[derive]:=(trainee TOP).sujet;
               IF
                CARD derives>rempli_max*(CAPACITY derives)
               THEN restructure_derives DONE;
               IF derive=sujet THEN
                UNTIL EMPTY candidats REPEAT
                 IF FULL file_attente THEN
                  restructure_attente
                 DONE;
                 derive FROM candidats;
                 file_attente APPEND derive
                REPETITION
               EXIT cree_descendants TAKE TRUE DONE;
               UNLESS derive IN terminal THEN
                candidats APPEND derive
               REPEAT cree_descendants DONE
              DONE
            REPETITION(*CYCLE traite_trainee*)
           REPETITION(*CYCLE cree_descendants*)
          DONE(*cas general*)
         DONE(*analyse*);
        /* /*EJECT*/ */
```

On remarque que la fonction *analyse* commence par vérifier si le mot qu'elle a chargé d'analyser n'a pas déjà été dérivé. Dans le cas contraire, elle prend à sa charge au moyen de la variable *candidats*, la queue *file_attente* et réinitialise cette dernière. On voit la manière dont on gère la pile *traînée* et la queue *candidats* pour produire des nouveaux dérivés. A un certain stade, il faut pouvoir arrêter la production de dérivés; c'est le rôle de la fonction *compatible*. Le résultat de cette fonction est faux si elle est à même de décider qu'il est impossible que le mot à analyser *sujet* puisse dériver du mot *objet* auquel elle est appliquée. Elle combine pour cela deux critères:

- Les membre droits de règles de production de la grammaire considérée sont au moins aussi long que leurs membre gauche. Il s'ensuit qu'un mot ne peut dériver d'un mot plus long que lui.

- Les règles de production ont une forme telle qu'une fois qu'un symbole terminal apparaît dans un mot, ce symbole subsistera dans tous les mots qui en sont dérivés. De plus, les parties du mot qui précède et suive ce symbole ne peuvent que s'allonger lors des dérivations successives. Soit donc un mot *n* de la forme *pxq* dans lequel *x* est un symbole terminal; un mot *m* ne peut dériver de ce mot *n* que si le terminal *x* y apparait dans sa sous-chaîne de longueur *1* + **length** *m* - **length** *n* débutant à la position *1* + **length** *p*.

```
descendant                          Vax Newton Compiler 0.2c14
                       Page    6
         Source listing


 1259     string VARIABLE ligne;
 1263     mot_terminal VARIABLE sujet
 1266  DO(*descendant*)
 1267    UNTIL end_file REPEAT
 1270      read(ligne);
 1275      WHILE terminal<>ligne REPEAT
 1280        IF
 1281          analyse((sujet:=terminal SPAN (ligne:=ligne ATLEFT terminal)))
 1297        THEN
 1298          print(line,#***Le mot "#,sujet,#" fait partie du langage***#);
 1309          imprime_derivation(sujet)
 1313        DEFAULT
 1314          print(line,"###Le mot "_#"#,sujet,
 1324                 #"# _"n'appartient pas au langage###",line)
 1330        DONE;
 1332        ligne:=terminal-ligne
 1337      REPETITION
 1338    REPETITION;
 1340    print(line,"<<<<<FIN DES ANALYSES>>>>>")
 1346  DONE(*descendant*)

    ****  No messages were issued  ****
```

Là, de nouveau, ce programme ne permet d'analyser en un temps raisonnable que des mots courts. On peut vérifier que l'explosion combinatoire intervient encore beaucoup plus vite si l'on ne tient compte que du critère de la longueur pour arrêter la production de dérivés: il n'est alors guère possible d'analyser des mots de plus de cinq ou six symboles.

Résultats :

```
***Le mot "x" fait partie du langage***
***Derivation du mot "x"***
 E T F B V x
<<<FIN>>>
```

###Le mot "x-y" n'appartient pas au langage###

Le mot "(x-y)" fait partie du langage
Derivation du mot "(x-y)"
E T F B (E) (EAT) (TAT) (FAT) (FAF) (F-F) (B-F) (V-F) (x-F)
(x-B) (x-V) (x-y)
<<<FIN>>>

Le mot "(y^y)" fait partie du langage
Derivation du mot "(y^y)"
E T F B (E) (T) (F) (B^F) (V^F) (y^F) (y^B) (y^V) (y^y)
<<<FIN>>>

Le mot "(-y^y)" fait partie du langage
Derivation du mot "(-y^y)"
E T F B (E) (AT) (AF) (-F) (-B^F) (-V^F) (-y^F) (-y^B) (-y^V)
(-y^y)
<<<FIN>>>

Le mot "f(x)" fait partie du langage
Derivation du mot "f(x)"
E T F B D(E) f(E) f(T) f(F) f(B) f(V) f(x)
<<<FIN>>>

Le mot "f(g(x))" fait partie du langage
Derivation du mot "f(g(x))"
E T F B D(E) f(E) f(T) f(F) f(B) f(D(E)) f(g(E)) f(g(T)) f(g(F))
f(g(B)) f(g(V)) f(g(x))
<<<FIN>>>

Le mot "f(x-y)" fait partie du langage
Derivation du mot "f(x-y)"
E T F B D(E) f(E) f(EAT) f(TAT) f(FAT) f(FAF) f(F-F) f(B-F)
f(V-F) f(x-F) f(x-B) f(x-V) f(x-y)
<<<FIN>>>

Le mot "(y*x)" fait partie du langage
Derivation du mot "(y*x)"
E T F B (E) (T) (TMF) (FMF) (BMF) (VMF) (yMF) (yMB) (yMV) (yMx)
(y*x)
<<<FIN>>>

Le mot "h(z)" fait partie du langage
Derivation du mot "h(z)"
E T F B D(E) h(E) h(T) h(F) h(B) h(V) h(z)
<<<FIN>>>

###Le mot "y/x" n'appartient pas au langage###

Le mot "z" fait partie du langage
Derivation du mot "z"
E T F B V z
<<<FIN>>>

<<<<<FIN DES ANALYSES>>>>>

*****Derives trouves*****

Mots terminaux engendres par la grammaire
(y^x) f(-x) f(-y) (y^z) f(-z) (x/y) (-y^z) f(g(z)) f(g(y)) h(z)
(x+y) h(y) f(g(x)) (-y^x) (-x) f(z) f((x)) f((y)) f(x*y) (y*x)
(z-y) f(x-y) f(+x) (y^y) (+y) f(+y) (y+x) f(x/y) (y+y) f((z)) y
(x*y) h(x) (y-y) f(y) x z (y/x) (y/y) (+x) (y*y) (-y^y) (x-x)
(x-y) f(x-z) f(-(x)) (x-z) f(x-x) (-z) f(x) f(+(x)) (-y) (y-x)
<<<FIN>>>

Mots derives auxiliaires
f(TMFMF) f(TMFMV) (D(E)AV) f(+V) (BAx) f(E+T) (TMF) (B-V) (TAz)
(TMV) (AT+y) (-y^V) f(B/V) (V-V) (B^FMV) f(AF) f(B-B^F) (y^(E))
(B/x) (-D(E)^F) f(D(E)MB) f(VMB^F) (-TAB) (B^FMB) (T+V) f(x/B)
(B+F) ((E)^F) (zAy) (FMV) (AT-y) (yMx) (T*F) (x^F) (-y^D(E))
(F-V) f(B/y) (V/V) (B) (E) (F) (yAF) f(g(B^F)) f(D(E)MV) (Az)
(F+T) (z-V) (T) (y/V) f(V-B^F) (T-y) f(F*F) f(V*F) f(g((E)))
(F+F) h(EAT) (z^F) (zMB) (+FAy) (EAB) (xMF) (x-(E)) (B*y) (z-B)
(yMz) (x-F) f(F+F) (Ay) (AFAy) (V-(E)) (F*x) (VM(E)) f(A(T))
D(ATAT) (E+T) (D(E)AB) f(-B^F) (FAT) h((E)) (D(E)My) (BAy) (x*V)
f(BMz) f(A(B)) (AT+F) f(TMB) (F*y) (TAF) (-FAT) (-TAy) (BAz)
f((E)-F) f(V-F) f(V) f(F-F) (V+B) f(-T) f(A(D(E))) f(AD(E)) h(B)
f(TAT) (E+V) (FAy) (zAB) (T+x) f(TMF) f((D(E))) f(f(E)) (-F)
(AT-F) (zMF) (E-x) (EAF) (EAy) (T-B^F) f(A(z)) (z-F) (y^D(E))
f(F/F) f(T) f(V/F) D(E+T) f(BMD(E)) (D(E)AT) (A(E)^F) (E-y)
(yAT) (VMV) f(A(AT)) (D(E)^F) f(A(E)) (BMF) (F+V) f(yMF) (B-F)
f(F*V) f(V*V) (E-z) (-B^F) (y^B^F) (V/x) f(x-F) f(T*B) f(A(EAT))
D(-T) (VA(E)) (AFAF) (V+F) D(E-T) (y^B) f((E)) f(EAF) (A(E))
f((B^F)) (V/y) f((E)My) f(F*y) f(V*y) (AT+T) f(-B) T f(FM(E))
(B/B) (BM(E)) ((E)Ay) (V*x) V f(x*F) (AT+V) f(B-F) (B-D(E))
f(x/F) B f(FAB^F) (D(E)-B) (AF-T) (yMB) ((E)AT) (T-B) ((E)AV)
(TMB^F) f(BAF) f((E)MF) f((E)MV) (V*y) ((E)AB) ((E)AF) (x/B)
(VAx) (zAF) (-V^B) (F-(E)) (B*B) f(x-D(E)) (FM(E)) f((B)) f(AV)
f(TMV) f(A((E))) (-(E)) f(B^FMV) f(TMFMB) (AT-V) f(x-(E)) (VAy)
(T-D(E)) (yM(E)) f(zMB) (E-(E)) (xAB^F) f(F/V) (zMx) f(V/V)
(y^F) (ABAT) (T-T) (E+B) (F*B) f(AT) f(T/B) f(g(EAT)) (T/V)
(y+V) f(T+T) (B/F) f(yMV) (zMy) (-TAF) (B+T) (x-B) f(FAB)
(AB^FAy) f(F/y) (E-F) f(V/y) f(x-V) (FAB) (yMF) (AV) (T-F)
(B^FMy) (AFAV) (y-F) (T-V) (T*V) f(yMy) (+B) (ATAB) f(D(T))
(y*V) f((F)) (x+B) f(xMz) (B+V) (AB^FAT) f(F-T) f(-D(E)) (E-B)
(D(E)MB) (BMx) (-TAV) f((TMF)) (EAB^F) f(x/V) (TAV) (E-T) (F/y)
f(-(T)) (B-x) (yAV) f(g(AT)) (TMF-T) (TMF-V) (E+F) (F*F) D(EAF)
(BMy) (AT-T) f(TMFAT) f(-(B)) (TMF-F) f(FMz) f(FAT) f(VMz) (B-y)
h(D(E)) (yA(E)) (TMF-y) (FMx) (-V) (Ax) (+F) (BMz) (-y^B^F)
(-(E)^F) (F-x) (-z^F) (TMFAV) (FAF) (y^V) (D(E)-T) (T+y)
f(B^FMy) h(F) (F-y) f(T*F) (TMFAB) (TMFAF) (FMy) (T-z) (D(E)Mx)
(B^FAT) (x*F) (V^B^F) (TMFAy) (TMFAx) (x+F) (E-D(E)) (AFAT)
(FMz) (ATAy) f(+(T)) (V*B) D(E) f(B^FMF) (B^FAy) (B^FAx) (F-z)
f(EAT) (V^B) (BMB^F) (D(E)Ax) f(BMx) f(-(E)) (BMD(E)) h(B^F)
f(xMx) (+TAB) (y/F) (V-x) f(A(y)) (FMD(E)) (VAB) ((E)Ax) (-TAT)
f(FMV) (+TAF) (D(E)MF) (-V^F) (VMy) (zMV) (VAB^F) (B^B^F)
(D(E)-F) f(B^FAF) (D(E)-V) f(ATAT) (V-D(E)) (xAx) f(D(AT)) (V/F)
f(A(B^F)) (yMy) f(TMD(E)) (V+T) f(FMx) f(VMx) f(h(E)) (BAD(E))
f(zMF) (xAF) (EAD(E)) (-T) (EATAx) (TMD(E)) g(E) (xAy) (B-(E))
(BMV) (FAx) f(TM(E)) (FAD(E)) f(D(EAT)) f(-(V)) f(y-F) (AB^F)
f(T/F) f((T)) (V+x) (E+y) (xAz) f(+(E)) (x-D(E)) f(FAF) f((E)MB)
(+TAy) (V+V) f(A(x)) (B^B) (TAT) (VMx) f(T*V) f(A(V)) (TAD(E))
(yAB^F) f(BAT) (FAz) f(((E))) (VAD(E)) (xAV) (-x^F) f(B*B)

```
 (yMD(E)) D(+T) (+FAT) (ATAF) (F/B) f((AT)) f((V)) f(FMB)
 f(xMB^F) f(VMB) f(BMB) f(BM(E)) (V-y) (xMB^F) f(xMD(E)) f(T*y)
 (x-B^F) (ATAD(E)) f(+B) (EAT-V) f(VMy) (B-B) (VMz) (EAT-B) (TMB)
 f(+(V)) (BAB^F) (zAT) f(Ay) (V-z) (y*F) (yAD(E)) (xAD(E))
 (EAT-y) (T+B) (TA(E)) (EATAT) (y+B) (EATAV) (FMB) (BMB) f(B-B)
 ((E)-y) (EATAB) (F-B) (EATAF) D(EAT) (V/B) (B^F) (AB^B^F) (y-T)
 (FAB^F) f(BMV) f(FMB^F) (+V) f(zMV) (zAV) (EATAy) f(x-B^F)
 ((E)-T) f(-V) ((E)-V) f(A(TMF)) (+B^F) (B/V) f(xMB) (EAT-F)
 ((E)-B) (F/F) ((E)-F) (F/x) f(T/V) (T/x) (B*x) (E-B^F) (-y^B)
 f(BMy) (yMV) (E+x) f(+F) f(zMy) (BAT) f(F-B^F) (y-V) (AF-y)
 f(A(F)) (x*B) (x/V) (F+B) f(-(F)) (B*V) (TMFAT) f(T/y) f(Az)
 (EAT-T) (T+F) (x-T) f(TMB^F) (ATAV) (y+F) (FMF) (y-B) (VMD(E))
 (-y^(E)) (T/y) (TMFMV) (xAB) (TMx) (F-F) (BAV) (TMFMB) (T-x)
 D(AF) (TMFMF) (BAF) f(D(E)-F) (AB^B) (F*V) f(D(E)MF) f(Ax)
 f(T-T) (EAT) (x/F) (TMFMy) (TMy) (xMV) (y*B) (B^FMF) (AF+T) h(V)
 D(TAT) (B+x) (x-V) f(B^FMB) f(F*B) f(g(T)) (AD(E)^F) E (T*x)
 (B^FMx) (TMF-B) (FAV) (TMz) (x+T) (yAx) f(TMz) f(+(F)) (B+y)
 h(TMF) f(g(TMF)) (+(E)) g(EAT) (VMB) (T*y) (EAV) (TAy) (-FAy)
 (V-B) (VMB^F) (yAy) D(T) (V-B^F) f(B*F) (xMx) (D(E)MV) (E-V)
 (ATAz) (-D(E)) f(D(E)My) f(D(E)) f(VMF) f(FMF) (F+y) (x+V) (yAz)
 (xA(E)) (V*F) (V-T) h(T) (xAT) (BAB) (xMy) f(F-B) (ABAy) (ATAT)
 f(F) (AV^F) (V^F) f(AB^F) (D(E)-y) (D(E)AF) h(E) (+TAV) (TAx)
 (xMz) (ATAB^F) f(FMD(E)) f(TMy) (TMFMx) D(EATAT) (A(EAT))
 (yMB^F) f(-F) f(F/B) f(VM(E)) (F-D(E)) f(z-F) f(E) (VMF) f(V*B)
 f(B^FAT) (EA(E)) (VAT) (V-F) (AF+y) (AF) f(B^F) f(xM(E)) (B/y)
 (EAx) f(TMx) f(g(B)) (V*V) f(xMF) (-B) f(B*V) (-B^B^F) (ATAx)
 (VAF) ((E)MV) (-y^F) f(B/F) (B-B^F) f(g(V)) f(g(D(E))) (z-T)
 (VAV) (AT+B) (B^FAV) (T-(E)) (T/B) (TM(E)) f(TAF) (y/B) (B^FAB)
 (EAz) f(BMF) (B^FAF) F f(V-B) (-V^B^F) (xM(E)) f(VMD(E)) D(AT)
 (AT) (A(T)) (FMB^F) h(AT) f(VMV) (F-B^F) f(x*B) (AD(E)) (+TAT)
 f(x*V) (VAz) (xMD(E)) (A(AT)) (T*B) g(AT) f(AB) (AT-B) (V+y)
 (B-z) f(B*y) f(F+T) f(xMy) ((E)Mx) ((E)My) (B+B) f(FMy) (BA(E))
 f(V/B) (F+x) f(B) f(B/B) (B*F) f(+T) (TAB) f(g(E)) (B-T)
 f(TMFMy) (yAB) f(EATAT) f((EAT)) (AB) (TAB^F) f(yMB) (zAx)
 f(+(B)) ((E)MB) (D(E)Ay) (ATA(E)) ((E)MF) f(E-T) (T/F) (-B^B)
 f(BMB^F) (F/V) (T+T) f(g(F)) f(x-B) (FA(E)) (y+T) (AFAB) (xMB)
 (+T) (F-T) f(xMV)
<<<FIN>>>

<<<<<FIN>>>>>
```

On va maintenant traiter une application où l'introduction d'une heuristique se révèle indispensable: il s'agit du problème du tour d'un cavalier sur un échiquier. On se donne un échiquier carré de dimension *ordre* = *8*. On cherche à faire parcourir à un cavalier, se déplaçant selon les règles du jeu d'échecs (chaque déplacement est un saut de deux cases horizontalement et d'une case verticalement ou vice-versa) toutes les cases du jeu en visitant une et une seule fois chaque case. Ce problème peut être programmé au moyen de la technique du retour arrière. Si l'on n'introduit pas d'heuristique, l'explosion combinatoire est inévitable. Pour l'apprécier, on suppose que le cavalier est sur l'une des deux cases à partir de laquelle un des coins de l'échiquier est accessible; on comprendra facilement qu'il est essentiel que le cavalier passe par le coin au coup suivant et en revienne par l'autre case (dans le cas contraire, le coin deviendrait une impasse). Si en l'absence d'heuristique, on fait un autre choix relativement tôt, le retour arrière devra nécessairement se faire jusqu'à ce point; entre temps, le cavalier aura dû explorer sans succès des quantités de chemins et l'on ne pourra trouver un parcours adéquat en un temps raisonnable.

Une heuristique, qui se révèle très satisfaisante, consiste à faire sauter le cavalier, en priorité, sur la case d'où il aura le moins d'issues possibles. On constate immédiatement que cette heuristique résoudra correctement le problème des coins. L'expérience montre qu'elle permet non seulement de trouver rapidement un parcours sur un échiquier d'ordre huit, mais qu'elle reste efficace sur les échiquiers nettement plus grands: elle permet encore de trouver un parcours sur un échiquier d'ordre vingt en un temps raisonnable.

La structure du programme *tour_cavalier* est donnée à la figure 36.

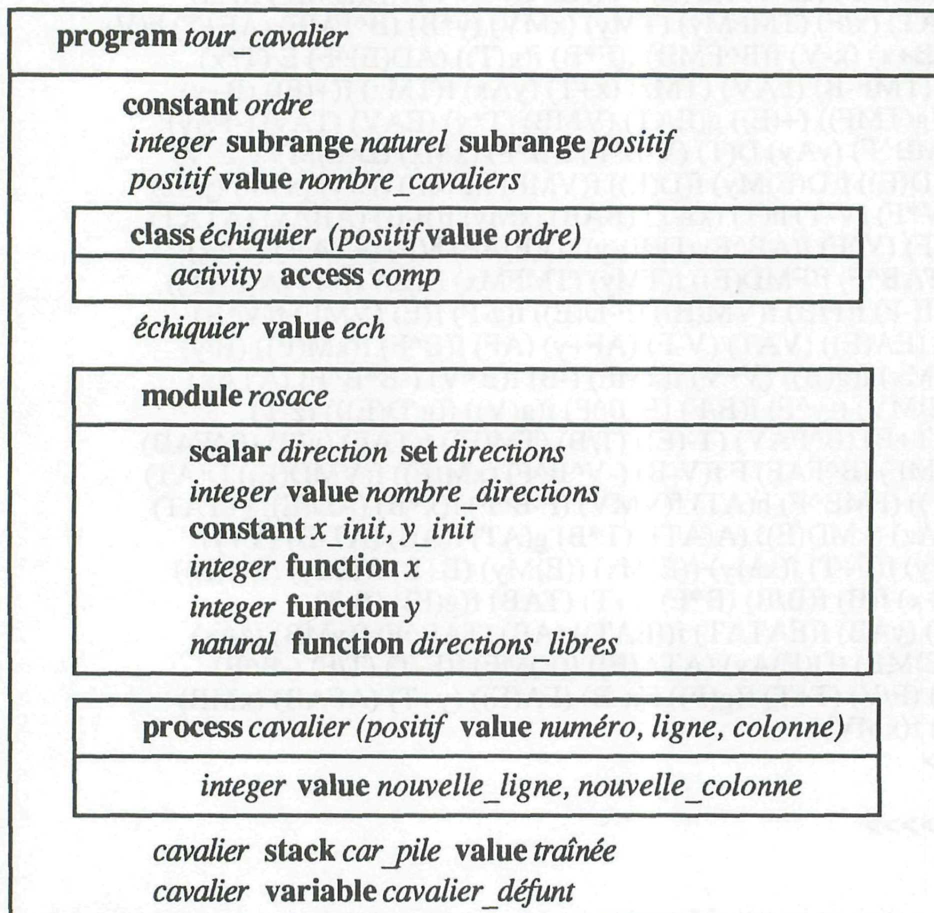


Figure 36

Les objets du type *échiquier* sont des échiquiers carrés dont la dimension est communiquée en paramètre lors de leur création. On accède aux cases de l'échiquier par l'intermédiaire de la fonction d'accès *comp* utilisée comme fonction d'indiçage. Les cases de l'échiquier sont des variables du type prédéfini *activity*; ce type joue le même rôle pour les types processus que le type *pointer* pour les types objets.

```
tour_cavalier                                  Vax Newton Compiler 0.2c9
                                           Page     1
                      Source listing

   1   /* /*OLDSOURCE=USER2:[RAPIN]TOUR_CAVALIER.NEW*/ */
   1   PROGRAM tour_cavalier DECLARE
   4     CONSTANT ordre=8(*dimension de l'echiquier*);
   9     integer SUBRANGE naturel(naturel>=0)
  17             SUBRANGE positif(positif>0)
  24             VALUE nombre_cavaliers=ordre**2;
  31
  31     CLASS echiquier
  33       INDEX comp  ATTRIBUTE ordre
  37       (positif VALUE ordre)
  42     (*Un echiquier carre de dimension  ordre  de cases contenant
  42       des activites. Initialement, les cases contiennent l'acti-
  42       vite vide NONE; les cases en dehors de l'echiquier peuvent
  42       etre consultees: elles contiennent une activite bidon non
  42       modifiable de l'exterieur
  42     *)
  42     DECLARE(*echiquier*)
  43       activity ROW ac_row VALUE tab=
  49         THROUGH ac_row(1 TO ordre**2):=NONE REPETITION;
  62       COROUTINE bord DO DONE;
  67       activity VARIABLE autres_cases:=bord;
  73
  73       activity ACCESS comp
  76         (integer VALUE j,k)
  83       (*Un acces a la case d'indices  j  et  k  de l'echiquier*)
  83       DO(*comp*)TAKE
  85         IF 0<j MIN k/\j MAX k<=ordre THEN
  98           tab[(PRED j)*ordre+k]
 109         DEFAULT
 110           autres_cases:=bord
 113         TAKE autres_cases DONE
 116       DONE(*comp*)
 117     DO(*echiquier*)DONE VALUE ech=echiquier(ordre);
 127   /* /*EJECT*/ */
```

Toute valeur d'un type processus peut être convertie, si le contexte l'exige, en une valeur du type *activity* (en sens contraire, la conversion nécessite un test d'exécution); les opérations valables sur un objet d'un type processus sont applicables aux valeurs du type *activity*: c'est notamment le cas de l'opérateur **state** et de l'énoncé **activate**. Il est aussi possible de leur appliquer les opérations disponibles sur les objets généraux du type *pointer*, en particulier l'énoncé **inspect** peut porter sur une valeur du type *activity*. Si le contexte l'exige, un objet processus du type *activity* peut être converti au type *pointer*; la conversion inverse n'est cependant pas disponible (même avec test à l'exécution). Ainsi, l'expression *pointer [***none***]* est admissible et sa valeur est l'objet vide **nil**; par contre l'expression *activity [***nil***]* n'est pas autorisée.

Le lecteur constatera que l'échiquier a été implanté comme un tableau non borné dont toutes les cases en dehors du carré de dimension *ordre* débouchent sur la même variable *autres_cases* dont la valeur est la coroutine *bord* Avec cette manière de faire, l'algorithme du cavalier pourra être programmé sans avoir à tester s'il se trouve sur une case près du bord: s'il tente de sauter sur une case en dehors de l'échiquier, cette case aura l'air occupée et il ne pourra donc pas s'y rendre.

```
MODULE rosace
  ATTRIBUTE direction,directions,nombre_directions,
            directions_libres,
            x,x_init,y,y_init
DECLARE(*rosace*)
  SCALAR direction(nne,ene,ese,sse,sso,oso,ono,nno)
     SET directions;
  (*Les directions possibles de deplacement du cavalier*)
  integer VALUE nombre_directions=SUCC ORD direction MAX;
  (*Le nombre de directions possibles*)

  CONSTANT x_init=1,y_init=1;
  (*Le point de depart du cavalier*)

  integer FUNCTION x
    (direction VALUE d)
  (*Le deplacement horizontal associe a la direction  d *)
  DO(*x*)TAKE
    CASE d WHEN
    oso,ono THEN -2|
    sso,nno THEN -1|
    sse,nne THEN 1|
    ese,ene THEN 2
    DONE
  DONE(*x*);

  integer FUNCTION y
    (direction VALUE d)
  (*Le deplacement vertical associe a la direction  d *)
  DO(*y*)TAKE
    CASE d WHEN
    nne,nno THEN 2|
    ene,ono THEN 1|
    ese,oso THEN -1|
    sse,sso THEN -2
    DONE
  DONE(*y*);

  naturel FUNCTION directions_libres
    (positif VALUE ligne,colonne)
  (*Le resultat est le nombre de cases libres atteignable, en
    un saut, par le cavalier a partir de la case donnee par
    ses numeros de ligne et de colonne.
  *)
  DECLARE naturel VARIABLE c:=0 DO
    FOR direction VALUE d REPEAT
      IF ech[ligne+y(d),colonne+x(d)]=NONE THEN
        c:=SUCC c
      DONE
    REPETITION
  TAKE c DONE(*directions_libres*)
DO(*rosace*)DONE;
```

```
tour_cavalier                                      Vax Newton Compiler 0.2c9
                                                   Page     3
                         Source listing

     /* /*EJECT*/ */
```

Le module *rosace* décrit, au moyen du type scalaire *direction* et des fonctions *x* et *y* les mouvements possibles du cavalier. Il y est défini le point initial du parcours au moyen de ses coordonnées *x_init* et *y_init*. La fonction *directions_libres*, utilisée pour implanter l'heuristique du cavalier, a pour résultat le nombre de cases vides que le cavalier peut atteindre depuis celle dont les coordonnées lui sont communiquées comme paramètres. On remarque que comme *y* représente un déplacement vertical, le résultat de cette fonction vient s'ajouter au numéro de la ligne où l'on suppose le cavalier; de même *x* dénote un déplacement horizontal et sera ajouté au numéro de colonne.

```
tour_cavalier                                      Vax Newton Compiler 0.2c9
                                                   Page     4
                         Source listing

      PROCESS cavalier
        VALUE moi
        ATTRIBUTE numero,ligne,colonne,nouvelle_ligne,nouvelle_colonne
        (positif VALUE numero,ligne,colonne)
      DECLARE(*cavalier*)
        directions ROW rangee_directions VALUE baquet=
          THROUGH
            rangee_directions(0 TO nombre_directions):=directions{}
          REPETITION;
        (*Cette rangee servira a etablir une heuristique des sauts
          possibles de ce cavalier
        *)

        integer VARIABLE lig VALUE nouvelle_ligne,
                         col VALUE nouvelle_colonne;
        directions REFERENCE bac
      DO(*cavalier*)
        ech[ligne,colonne]:=moi
      RETURN
        FOR direction VALUE d REPEAT
          IF ech[(lig:=ligne+y(d)),(col:=colonne+x(d))]=NONE THEN
            bac->baquet[directions_libres(lig,col)]:=bac+d
          DONE
        REPETITION;
        THROUGH baquet VALUE bac REPEAT
          THROUGH bac VALUE dir REPEAT
            lig:=ligne+y(dir); col:=colonne+x(dir)
          RETURN REPETITION
        REPETITION;
        ech[ligne,colonne]:=NONE
      DONE(*cavalier*)STACK cav_pile VALUE trainee=
        (cav_pile(nombre_cavaliers) PUSH cavalier(1,x_init,y_init));

      cavalier VARIABLE cavalier_defunt
     /* /*EJECT*/ */
```

Le type processus *cavalier* décrit l'algorithme du cavalier; plus spécifiquement, il est créé un cavalier pour chaque case du parcours. Au début de sa partie exécutable, le cavalier se place sur la case qui lui a été fournie comme paramètre; à ce sujet, il aurait été possible de remplacer l'assignation *ech [ligne, colonne]:=moi* par *ech [ligne, colonne]:=***current** : en-effet **current** dénote toujours la coroutine en train d'élaborer le code correspondant. Après ce placement, le cavalier se détache. Chaque fois qu'on le réactive, il va proposer une case libre dans laquelle il pourrait sauter en tenant compte de l'heuristique envisagée. On remarque que celle-ci est programmée au moyen de la rangée *baquet*; étant donné un indice entier *k* inclus entre *0* et *nombre_directions*, l'élément *baquet [k]* a pour valeur l'ensemble des directions de saut aboutissant à une case libre à partir de laquelle le cavalier peut atteindre *k* cases libres. Il n'est pas difficile d'établir cette rangée. Ensuite, pour établir l'heuristique souhaitée, il suffit de considérer, dans l'ordre croissant de leurs indices, les éléments de la rangée *baquet* puis, pour chacun d'entre eux, de proposer les déplacements dont les directions sont incluses dans l'ensemble correspondant. C'est au moyen des attributs *nouvelle_ligne* et *nouvelle_colonne* qu'il est possible de consulter le déplacement proposé. Une fois que le cavalier n'a plus de déplacement à proposer, il achève son exécution en libérant la case qu'il avait occupée. En-effet si on réactive le cavalier alors qu'il n'a plus de déplacement à proposer, cela signifie qu'une impasse a été atteinte et qu'un retour arrière va s'imposer: il faut donc bien rétablir la situation antérieure; en particulier la case du cavalier doit être libre.
On remarque que la traînée est une pile de cavaliers; dans le cas présent, il est clair qu'elle ne peut contenir plus de *nombre_cavaliers = ordre* ** 2 éléments. Initialement, on y place le premier cavalier.

```
tour_cavalier                                    Vax Newton Compiler 0.2c9
                                          Page     5
               Source listing

DO(*tour_cavalier*)
  CYCLE cherche_tour REPEAT

    ACTIVATE trainee TOP NOW;

  IF STATE trainee TOP=terminated THEN
    cavalier_defunt POP trainee;

    IF EMPTY trainee THEN
      print(line,"###Tour cavalier impossible###")
    EXIT cherche_tour DONE

  REPEAT cherche_tour DONE;

    CONNECT trainee TOP THEN
      trainee PUSH cavalier(SUCC numero,nouvelle_ligne,nouvelle_colonne)
    DONE;

  IF (trainee TOP).numero=nombre_cavaliers THEN
    print(line,"***Tour cavalier possible***");
    FOR integer VALUE j FROM ordre TO 1 REPEAT
      line;
      FOR integer VALUE k FROM 1 TO ordre REPEAT
        INSPECT ech[j,k] WHEN
        cavalier THEN edit(numero,4,0)
        DEFAULT print(_"???") DONE
      REPETITION
    REPETITION;
    print(line,"<<<FIN>>>")
  EXIT cherche_tour DONE;

  REPETITION(*CYCLE cherche_tour*)
DONE(*tour_cavalier*)

  ****  No messages were issued  ****
```

La partie exécutable du programme incorpore l'algorithme de recherche de parcours par retours arrière. Le lecteur peut constater la manière dont elle crée des nouveaux cavaliers et les incorpore à la traînée, ainsi que celle dont elle en retire les cavaliers qui n'ont plus d'usage. Evidemment, dès que l'on a pu placer un cavalier de numéro égal à *nombre_cavaliers*, cela implique que l'on a obtenu un parcours complet. Il suffit d'imprimer ce dernier en parcourant, ligne par ligne, les cases de l'échiquier et en imprimant le numéro du cavalier qui s'y trouve.

Résultat

```
***Tour cavalier possible***
  22   7  44  39  24   9  28  63
  43  40  23   8  45  62  25  10
   6  21  42  59  38  27  64  29
  41  58  37  46  61  54  11  26
  20   5  60  53  36  47  30  51
  57   2  35  48  55  52  15  12
   4  19  56  33  14  17  50  31
   1  34   3  18  49  32  13  16
<<<FIN>>>
```

Une variante, plus difficile, consiste à rendre le tour réentrant; le tour sera réentrant si, à partir de la dernière case visitée, le cavalier peut rejoindre, en un seul saut, sa case initiale. On constate que le tour obtenu par le programme précédent n'est pas réentrant.

On va donner une version modifiée de ce programme; celle-ci ne cherche que les tours réentrants. Elle a permis d'en obtenir un, en un temps encore raisonnable, pour un échiquier d'ordre huit. Par contre, pour un échiquier d'ordre vingt, elle se heurte à l'explosion combinatoire. (On notera qu'un tour réentrant n'est possible que sur un échiquier d'ordre pair).

```
tour_cavalier                                      Vax Newton Compiler 0.2c9
                                     Page    1
                    Source listing

   /* /*OLDSOURCE=USER2:[RAPIN]TOUR_CAVALE.NEW*/ */
   PROGRAM tour_cavalier DECLARE
     CONSTANT ordre=8 (*dimension de l'echiquier*);
     integer SUBRANGE naturel(naturel>=0)
             SUBRANGE positif(positif>0)
             VALUE nombre_cavaliers=ordre**2;

     CLASS echiquier
       INDEX comp  ATTRIBUTE ordre
       (positif VALUE ordre)
     (*Un echiquier carre de dimension  ordre  de cases contenant
       des activites. Initialement, les cases contiennent l'acti-
       vite vide NONE; les cases en dehors de l'echiquier peuvent
       etre consultees: elles contiennent une activite bidon non
       modifiable de l'exterieur
     *)
   /* /*NO_LIST*/ */
   /* /*LIST*/ */
   MODULE rosace
     ATTRIBUTE direction,directions,nombre_directions,
               directions_libres,contigu,
               x,x_init,y,y_init
   /* /*NO_LIST*/ */
     /* /*LIST*/ */
     Boolean FUNCTION contigu
       (positif VALUE ligne,colonne)
     (*Vrai ssi la case donnee par ses numeros de ligne et de
       colonne est contigue a la case de depart du cavalier
     *)
     DECLARE direction VARIABLE d:=direction MIN DO
       TAKE CYCLE examen REPEAT

       IF
         ligne+y(d)=y_init/\colonne+x(d)=x_init
       EXIT examen TAKE TRUE DONE;

       IF
         d=direction MAX
       EXIT examen TAKE FALSE DONE;

         d:=SUCC d
       REPETITION(*TAKE CYCLE examen*)
     DONE(*contigu*)
   DO(*rosace*)DONE;
   /* /*EJECT*/ */
```

Ce nouveau programme ne sera pas commenté en détail; dans le module *rosace* on a ajouté la fonction *contigu* qui permet de vérifier si la case donnée comme paramètre est contiguë à la case initiale du cavalier.

```
tour_cavalier                                   Vax Newton Compiler 0.2c9
                                           Page     2
                 Source listing


   PROCESS cavalier
     VALUE moi
     ATTRIBUTE numero,ligne,colonne,nouvelle_ligne,nouvelle_colonne
     (positif VALUE numero,ligne,colonne)
   DECLARE(*cavalier*)
     directions ROW rangee_directions VALUE baquet=
       THROUGH
         rangee_directions(0 TO nombre_directions):=directions{}
       REPETITION;
     (*Cette rangee servira a etablir une heuristique des sauts
       possibles de ce cavalier
     *)

     integer VARIABLE lig VALUE nouvelle_ligne,
                      col VALUE nouvelle_colonne;
     directions REFERENCE bac
   DO(*cavalier*)
     ech[ligne,colonne]:=moi
   RETURN
     IF numero<nombre_cavaliers THEN
       FOR direction VALUE d REPEAT
         lig:=ligne+y(d); col:=colonne+x(d);
         IF ech[lig,col]=NONE THEN
           bac->UNLESS contigu(lig,col) THEN
                  baquet[directions_libres(lig,col)]
                DEFAULT
                  baquet[nombre_directions]
                DONE
              :=bac+d
         DONE
       REPETITION;
       THROUGH baquet VALUE bac REPEAT
         THROUGH bac VALUE dir REPEAT
           lig:=ligne+y(dir); col:=colonne+x(dir)
         RETURN REPETITION
       REPETITION
     DEFAULT
       IF contigu(ligne,colonne) THEN
         lig:=y_init; col:=x_init
       RETURN DONE
     DONE;
     ech[ligne,colonne]:=NONE
   DONE(*cavalier*)STACK cav_pile VALUE trainee=
     (cav_pile(nombre_cavaliers) PUSH cavalier(1,x_init,y_init));
 /* /*EJECT*/ */
```

Dans le type processus *cavalier*, on a complété l'heuristique par le point suivant: lorsqu'un cavalier peut sauter à une case d'où il peut rejoindre sa case initiale, ce saut ne sera proposé qu'en dernier lieu afin d'éviter d'occuper prématurément la (ou les) case(s) qui permettraient de rendre le tour réentrant si le dernier cavalier venait à s'y installer.

```
tour_cavalier                                   Vax Newton Compiler 0.2c9
                                      Page    3
                     Source listing

   cavalier VARIABLE cavalier_defunt
 DO(*tour_cavalier*)
   CYCLE cherche_tour REPEAT

     ACTIVATE trainee TOP NOW;

   IF STATE trainee TOP=terminated THEN
     cavalier_defunt POP trainee;
     IF EMPTY trainee THEN
       print(line,"###Tour reentrant impossible###")
     EXIT cherche_tour DONE

   REPEAT cherche_tour DONE;

   IF (trainee TOP).numero=nombre_cavaliers THEN
     print(line,"***Tour reentrant possible***");
     FOR integer VALUE ligne FROM ordre TO 1 REPEAT
       line;
       FOR integer VALUE colonne FROM 1 TO ordre REPEAT
         INSPECT ech[ligne,colonne] WHEN
         cavalier THEN edit(numero,4,0)
         DEFAULT print(_"???") DONE
       REPETITION
     REPETITION;
     print(line,"<<<FIN>>>")
   EXIT cherche_tour DONE;

     CONNECT trainee TOP THEN
       trainee PUSH cavalier(SUCC numero,nouvelle_ligne,nouvelle_colonne)
     DONE
   REPETITION
 DONE(*tour_cavalier*)
```

Résultat:

```
***Tour reentrant possible***
  22   7  44  39  24   9  28  59
  43  40  23   8  45  60  25  10
   6  21  42  61  38  27  58  29
  41  62  37  46  57  54  11  26
  20   5  56  53  36  47  30  51
  63   2  35  48  55  52  15  12
   4  19  64  33  14  17  50  31
   1  34   3  18  49  32  13  16
<<<FIN>>>
```

Si l'on compare le tour réentrant obtenu avec le premier tour, on remarque qu'il n'en diffère que vers la fin, plus spécifiquement à partir du cavalier *56*, ce qui explique que ce tour réentrant a pu être obtenu facilement.

Parfois, il ne suffit pas d'obtenir une solution d'un problème résoluble par une méthode de retour arrière. On aimerait les obtenir toutes ou, en tout cas, produire un algorithme capable de les produire une à une. Dans un tel cas, après production d'une solution, il faudra prévoir un retour arrière, comme si l'on avait abouti à une impasse, et poursuivre l'algorithme. Ce dernier ne s'arrêtera que si la traînée devient vide, signifiant qu'il n'y a plus de variantes à explorer. A titre d'exemple, on considère le réseau de la figure 37.

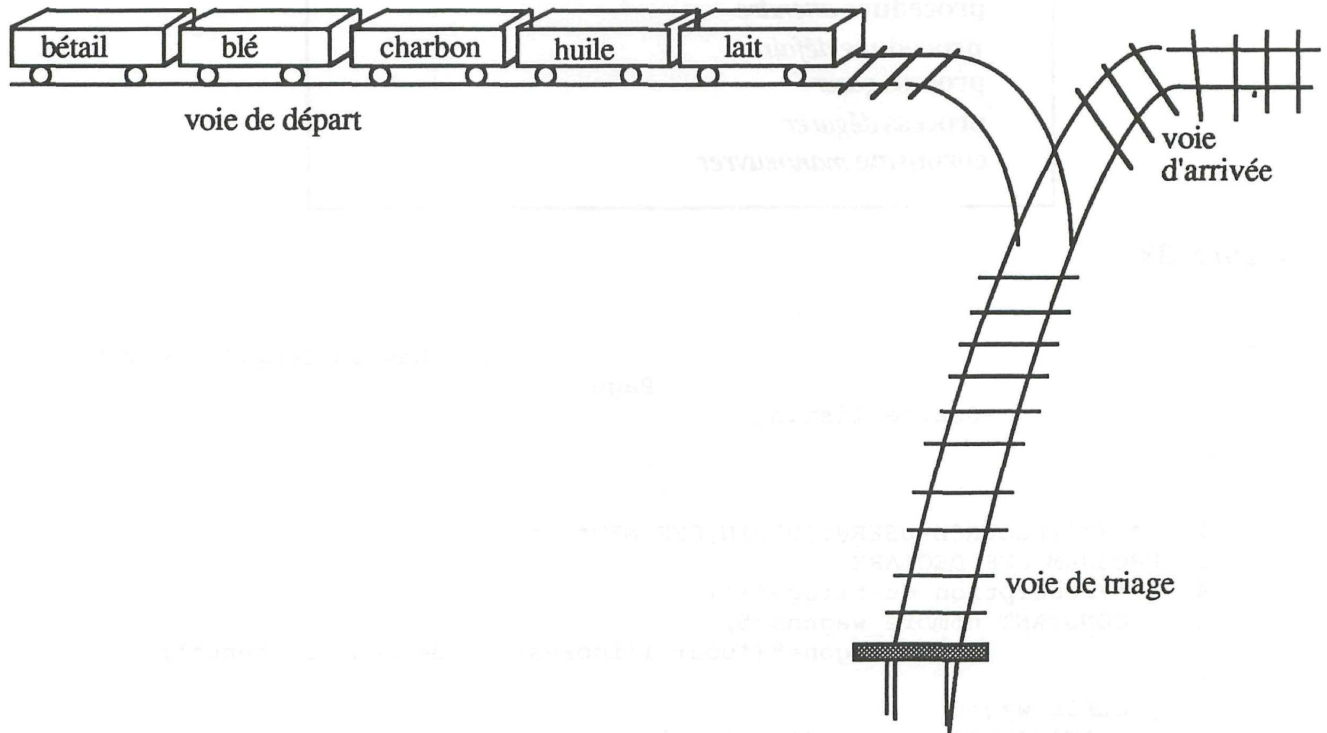


Figure 37

On veut savoir les trains qu'il est possible d'obtenir sur la voie d'arrivée sous l'hypothèse que seules les manoeuvres suivantes sont autorisées:

- Déplacer un wagon de la voie de départ sur la voie de triage.
- Déplacer un wagon de la voie de triage sur la voie d'arrivée.

On suppose donc qu'il est interdit de rétrograder un wagon de la voie d'arrivée à la voie de triage ou de la voie de triage à la voie de départ. On suppose par contre que la voie de triage est assez longue pour contenir les cinq wagons.

Ceci est réalisé dans le programme *CFF* de structure donnée à la figure 38.

program *CFF*

constant *nombre_wagons*

class *wagon (string* **value** *marchandise)*

procedure *imprimer*

wagon **stack** *voie* **value** *départ, triage, arrivée*
activity **stack** *programme* **value** *manoeuvres*
procedure *attendre*
procedure *défaire*
process *garer*
process *dégarer*
coroutine *manoeuvrer*

Figure 38

```
CFF                                            Vax Newton Compiler 0.2c9
                                          Page     1
                Source listing


  /* /*OLDSOURCE=USER@:[RAPIN]CFF.NEW*/ */
  PROGRAM CFF DECLARE
  (***Descrption du triage***)
    CONSTANT nombre_wagons=5,
             champ_wagon=8(*pour l'impression de leur contenu*);

    CLASS wagon
      ATTRIBUTE marchandise,imprimer
      (string VALUE marchandise)
    DECLARE
      PROCEDURE imprimer(integer VALUE champ) DO
      (*imprimer la marchandise avec  champ  caracteres*)
        print(marchandise); space(champ-LENGTH marchandise)
      DONE(*imprimer*)
    DO(*wagon*)DONE;

    wagon STACK voie VALUE
      depart=(print("***Train initial***",line);
              THROUGH (voie(nombre_wagons)
                       PUSH wagon("Betail")
                       PUSH wagon("Ble")
                       PUSH wagon("Charbon")
                       PUSH wagon("Huile")
                       PUSH wagon("Lait"))
                      VALUE wag
              REPEAT wag.imprimer(champ_wagon) REPETITION),
      triage= (line;
              voie(nombre_wagons)),
      arrivee=voie(nombre_wagons);
  /* /*EJECT*/ */
```

Le début du programme a une structure très simple; on constate que les trois voies ont été définies sous la forme de piles aptes à contenir les *nombre_wagons = 5* wagons considérés. La voie *départ* est initialisée avec l'ensemble des wagons; leur contenu est imprimé lors de cette opération pour illustrer le train initial.

```
CFF                                              Vax Newton Compiler 0.2c9
                                          Page     2
                  Source listing

   125  (***Description des operations de manoeuvres***)
   125   activity STACK programme VALUE manoeuvres=
   131   (*pour les manoeuvres planifieees ulterieurement*)
   131     (programme(3*nombre_wagons) PUSH CURRENT);
   142
   142   PROCEDURE attendre DO
   145   (*Planifie plus tard le reste de la manoeuvre courante*)
   145     manoeuvres PUSH CURRENT
   148   RETURN DONE;
   151
   151   PROCEDURE defaire DECLARE
   154   (*Backtrack: elimine l'effet de la manoeuvre courante et
   154     amorce la prochaine manoeuvre prevue.
   154   *)
   154     activity VARIABLE manoeuvre
   157   DO(*defaire*)
   158     WHILE
   159       manoeuvre POP manoeuvres;
   163       ACTIVATE manoeuvre NOW
   166     TAKE STATE manoeuvre=terminated REPETITION
   172   DONE(*defaire*);
   174
   174   PROCESS garer DECLARE
   177   (*manoeuvre un wagon dans le triage, avec backtrack*)
   177     wagon VARIABLE wag
   180   DO RETURN
   182     wag POP depart; triage PUSH wag; attendre;
   192     (*Backtrack: restitue l'etat anterieur*)
   192     wag POP triage; depart PUSH wag
   199   DONE(*garer*);
   201
   201   PROCESS degarer DECLARE
   204   (*manoeuvre un wagon dans le  nouveau train, avec backtrack*)
   204     wagon VARIABLE wag
   207   DO RETURN
   209     wag POP triage; arrivee PUSH wag; attendre;
   219     (*Backtrack: restitue l'etat anterieur*)
   219     wag POP arrivee; triage PUSH wag
   226   DONE(*degarer*);
   228  /* /*EJECT*/ */
```

La pile *manoeuvres* représente la traînée; il y est inséré initialement le programme principal (qui peut être considéré comme une coroutine mise en oeuvre par le système d'exploitation); la raison de cette insertion apparaîtra plus loin.

Dans la pile *manoeuvres*, il est essentiellement placé des coroutines d'un des types processus *garer* ou *dégarer*. Il s'agit d'ailleurs de processus très voisins: un processus *garer* fait passer un wagon de la voie de départ au triage, puis en cas de réactivation le remet sur la voie de départ;

de même un processus *dégarer* fait passer un wagon de la voie de triage à la voie d'arrivée avec retour arrière possible de la voie d'arrivée au triage. Les coroutines de ces deux types de processus ont chacune deux points d'arrêt: le premier immédiatement après la création de l'objet correspondant et le second, après l'accomplissement de la manoeuvre requise, intervient par l'intermédiaire de la procédure *attendre*.

Le retour arrière sera commandé par une application de la procédure *défaire* . Cette procédure prend un à un les éléments de la traînée et les réactive jusqu'à ce qu'elle en trouve un dont cette réactivation ne débouchera pas sur l'achèvement de son exécution. Un tel élément sera normalement un processus *garer* ou *dégarer* arrêté sur son **return** initial: il représentera une variante de manoeuvre qui avait été prévue, mais qui n'avait pas été encore été mise en oeuvre. Par contre, les processus dont la réactivation par la procédure *défaire* a conduit à l'achèvement de leur exécution seront normalement des processus *garer* ou *dégarer* arrêté sur leur point d'arrêt intermédiaire; leur réactivation leur fera faire la manoeuvre inverse de celle qu'ils ont accomplie initialement afin de pouvoir revenir à un état antérieur du système et d'en explorer une autre variante de manoeuvre.

C'est cette conception du retour arrière qui a conduit à stocker au fonds de la traînée le programme principal. Il servira de sentinelle lorsqu'il n'y a plus de variante de manoeuvre à explorer; pour cela, et comme les processus *garer* et *dégarer*, la partie exécutable du programme *CFF* sera dotée de deux points d'arrêt.

```
CFF                                              Vax Newton Compiler 0.2c9
                                   Page    3
                 Source listing

  (***La partie active du programme***)
  COROUTINE manoeuvrer DO
    RETURN
    print("***Liste des trains constructibles***",line);
    UNTIL
      UNTIL
        IF EMPTY triage THEN
          ACTIVATE garer NOW ELSE
        IF EMPTY depart THEN
          ACTIVATE degarer NOW
        DEFAULT(*deux manoeuvres sont possibles*)
          manoeuvres PUSH degarer(*pour plus tard*);
          ACTIVATE garer NOW
        DONE
      TAKE FULL arrivee REPETITION;
      (*un train a ete forme*)
      THROUGH arrivee VALUE wag REPEAT
        wag.imprimer(champ_wagon)
      REPETITION;
      line;
      (*va charcher manoeuvre suivante*) defaire
    TAKE EMPTY manoeuvres REPETITION;
    print("***FIN***")
  DONE(*manoeuvrer*)

  DO(*CFF*)
    manoeuvrer RESUME
  RETURN(*CFF*)DONE

  ****  No messages were issued  ****
```

Le gros de la structure de contrôle requise a été placé dans la coroutine *manoeuvrer*, celle-ci est initialement détachée. Au début de la partie exécutable du programme *CFF*, on remarque la clause *manoeuvrer* **resume**. Syntaxiquement, le symbole **resume** peut séparer deux énoncés, tout comme le point-virgule ou le **return**. Le séparateur **resume** doit être précédé d'un énoncé qui produit pour résultat une coroutine détachée; l'effet est de réactiver cette dernière (tout comme si elle était l'objet d'un énoncé **activate**) en lieu et place de la coroutine qui exécute cette clause qui se détache (comme s'elle trouvait-là un **return**). La clause *manoeuvre* **resume** est donc le premier point d'arrêt du programme principal; le second suit immédiatement cette clause.

La partie exécutable de la coroutine *manoeuvre* est relativement simple à comprendre. Si l'une des voies *départ* ou *triage* est vide mais pas l'autre, il n'y a pas de choix: on amorce la manoeuvre requise en créant une coroutine du type approprié et en l'activant immédiatement. Si ces deux voies contiennent des wagons, il y a deux variantes de manoeuvres. On choisit arbitrairement celle qui sera exécuté en premier (on a choisi *garer*); avant de la créer et de l'activer, on a créé l'objet relatif à l'autre manoeuvre possible (donc *dégarer* vu le choix précédent) et on l'a laissé, arrêté sur son premier **return**, sur la traînée.

On remarque que dans ce programme, le retour arrière n'intervient qu'après la production d'une solution. Il n'y a en-effet pas à proprement parler d'impasses (si on a commencé une manoeuvre, il est toujours possible de la compléter pour aboutir à un train sur la voie d'arrivée).

```
***Train initial***
Betail  Ble     Charbon Huile   Lait
***Liste des trains constructibles***
Betail  Ble     Charbon Huile   Lait
Ble     Betail  Charbon Huile   Lait
Ble     Charbon Betail  Huile   Lait
Ble     Charbon Huile   Betail  Lait
Ble     Charbon Huile   Lait    Betail
Charbon Betail  Ble     Huile   Lait
Charbon Ble     Betail  Huile   Lait
Charbon Ble     Huile   Betail  Lait
Charbon Ble     Huile   Lait    Betail
Charbon Huile   Betail  Ble     Lait
Charbon Huile   Ble     Betail  Lait
Charbon Huile   Ble     Lait    Betail
Charbon Huile   Lait    Betail  Ble
Charbon Huile   Lait    Ble     Betail
Huile   Betail  Ble     Charbon Lait
Huile   Ble     Betail  Charbon Lait
Huile   Ble     Charbon Betail  Lait
Huile   Ble     Charbon Lait    Betail
Huile   Charbon Betail  Ble     Lait
Huile   Charbon Ble     Betail  Lait
Huile   Charbon Ble     Lait    Betail
Huile   Charbon Lait    Betail  Ble
Huile   Charbon Lait    Ble     Betail
Huile   Lait    Betail  Ble     Charbon
Huile   Lait    Ble     Betail  Charbon
Huile   Lait    Ble     Charbon Betail
Huile   Lait    Charbon Betail  Ble
Huile   Lait    Charbon Ble     Betail
Lait    Betail  Ble     Charbon Huile
Lait    Ble     Betail  Charbon Huile
Lait    Ble     Charbon Betail  Huile
Lait    Ble     Charbon Huile   Betail
Lait    Charbon Betail  Ble     Huile
Lait    Charbon Ble     Betail  Huile
Lait    Charbon Ble     Huile   Betail
Lait    Charbon Huile   Betail  Ble
Lait    Charbon Huile   Ble     Betail
Lait    Huile   Betail  Ble     Charbon
Lait    Huile   Ble     Betail  Charbon
Lait    Huile   Ble     Charbon Betail
Lait    Huile   Charbon Betail  Ble
Lait    Huile   Charbon Ble     Betail
***FIN***
```

On remarque qu'il a été possible de former *42* trains distincts sur la voie d'arrivée; on peut montrer, par récurrence, que si le nombre de wagons à manoeuvrer selon cette règle du jeu était égal à *n*, il serait possible de former $(2 * n)! / (n! * (n + 1)!)$ trains distincts sur la voie d'arrivée.
On peut se demander pourquoi la capacité de la traînée a été fixée à *3 * nombre_wagons*. On peut tout de suite remarquer qu'au moment où on aboutit à un train sur la voie d'arrivée, il y a en tout cas sur la traînée pour chaque wagon un processus *garer* et un processus *dégarer* arrêtés sur leur **return** intermédiaire qui permet de remettre, lors du retour arrière, le wagon à sa place initiale. Une capacité de *2 * nombre-wagons* est donc en tout cas nécessaire. Elle ne suffit cependant pas puisque la traînée pourra contenir, en plus, des variantes de manoeuvres planifiées, mais non encore explorées. Considérant le cas où l'on a placé les *nombre_wagons* wagons sur le triage avant de tous les sortir sur la voie d'arrivée, on constate qu'en *nombre_wagons-1* instants, un autre choix de manoeuvre (*dégarer*) était possible et a dû être empilé sur la traînée. Tenant compte que le programme principal doit aussi avoir sa place sur la traînée, cette dernière doit bien avoir une capacité de *3 * nombre_wagons*.

Chapitre 12

Queues de priorité

Une queue de priorité est, par définition un ensemble de données dans lequel il est possible, en tout temps, d'insérer de nouveaux éléments; les éléments d'une queue de priorité en sont retirés selon un critère de priorité fixé à l'avance.

L'interface d'une queue de priorité de valeurs réelles peut, par exemple, être définie au moyen de la classe protocole *queue_priorité* du module *classage* suivant.

```
module classage
    attribute queue_priorité, relation,
        constructeur, sélecteur, destructeur
declare (*classage*)

    Boolean functor relation (real value gauche, droite);
    actor constructeur (real value val);
    real functor selecteur;
    actor destructeur (real reference var);

    class queue_priorité
        value moi
        attribute prioritaire, vide, longueur,
            introduire, premier, retirer
        (relation value prioritaire;
         constructeur value cons;
         selecteur value prem;
         destructeur value des)

    (* Un objet de la classe queue_priorité est une queue de valeurs réelles ordonnées
       selon la relation prioritaire ; cette dernière satisfera aux contraintes suivantes
       pour tout ensemble de valeurs réelles x, y et z :

        prioritaire [x, y] nand prioritaire [y, x]
        prioritaire [x, y] nor prioritaire [x, y ] |>
            (prioritaire [z, x] == prioritaire [z, y])
        proritaire [x, y] /\prioritaire [y, z] |>
            prioritaire [x, z]

     L'implanteur fournira les objets procéduraux suivants:

        cons[x]     insère dans la queue un nouvel élément de valeur x
        prem        livre la valeur de l'élément prioritaire
        des [v]     élimine de la queue son élément prioritaire et en stocke la
                    valeur dans la variable v
    *)

    declare (*classage*)
        integer variable compte value longueur := 0;

        (*Le nombre d'éléments de la queue concernée*)
        Boolean expression vide =
        (* Vrai ssi la queue est vide*)
```

```
        longueur = 0;
queue_priorité function introduire
        (real value val)
(* Place dans la queue un nouvel élément de valeur val. Le résultat est
 la queue concernée.
*)
do compte := succ compte; cons[val] take moi done;
real expression premier =
(* La valeur de l'élément prioritaire.
   condition d'emploi: ~ vide
*)
        ( if vide then
           print ("### premier porte sur une queue vide ###", line)
         return done;
         prem eval);
queue_priorité function retirer
        (real reference var)
(*Retire de la queue son élément prioritaire et en place la valeur dans
   la variable désignée par var. Le résultat est la queue concernée.
  condition d'emploi: ~ vide
*)
do (* retirer*)
        if vide then
           print ("###retirer appliqué à une queue vide ###", line)
        return done;
        des [ var]; compte : = pred compte
take moi done (*retirer*)

    do (*queue_priorité*) done
do (*classage*) done
```

A priori, une queue de priorité pourrait être implantée au moyen d'une liste triée; ceci présente cependant l'inconvénient que le constructeur sera en général lent: il exige le parcours de la moitié de la liste, en moyenne, pour trouver le point où l'insertion doit avoir lieu. Il n'est de plus pas possible, en général, de supposer que la queue sera construite en une fois, ce qui permettrait d'adapter la méthode vue dans le programme *piliste* du chapitre 10.

On va montrer deux implantations possibles de queues de priorité; toutes deux sont basées sur la notion d'arbre de priorité. Un arbre de priorité est un arbre binaire; la valeur stockée à chaque noeud d'un arbre de priorité sera prioritaire (ou eventuellement de même priorité) par rapport aux valeurs stockéees dans chacun de ses deux sous-arbres.

Exemples:

Soit la déclaration:

```
relation value inf =
    body relation (real value gauche, droite) do
    take x<y done
```

L'arbre de la figure 38 est un arbre de priorité ordonné selon la relation *inf* mais non celui de la figure 39.

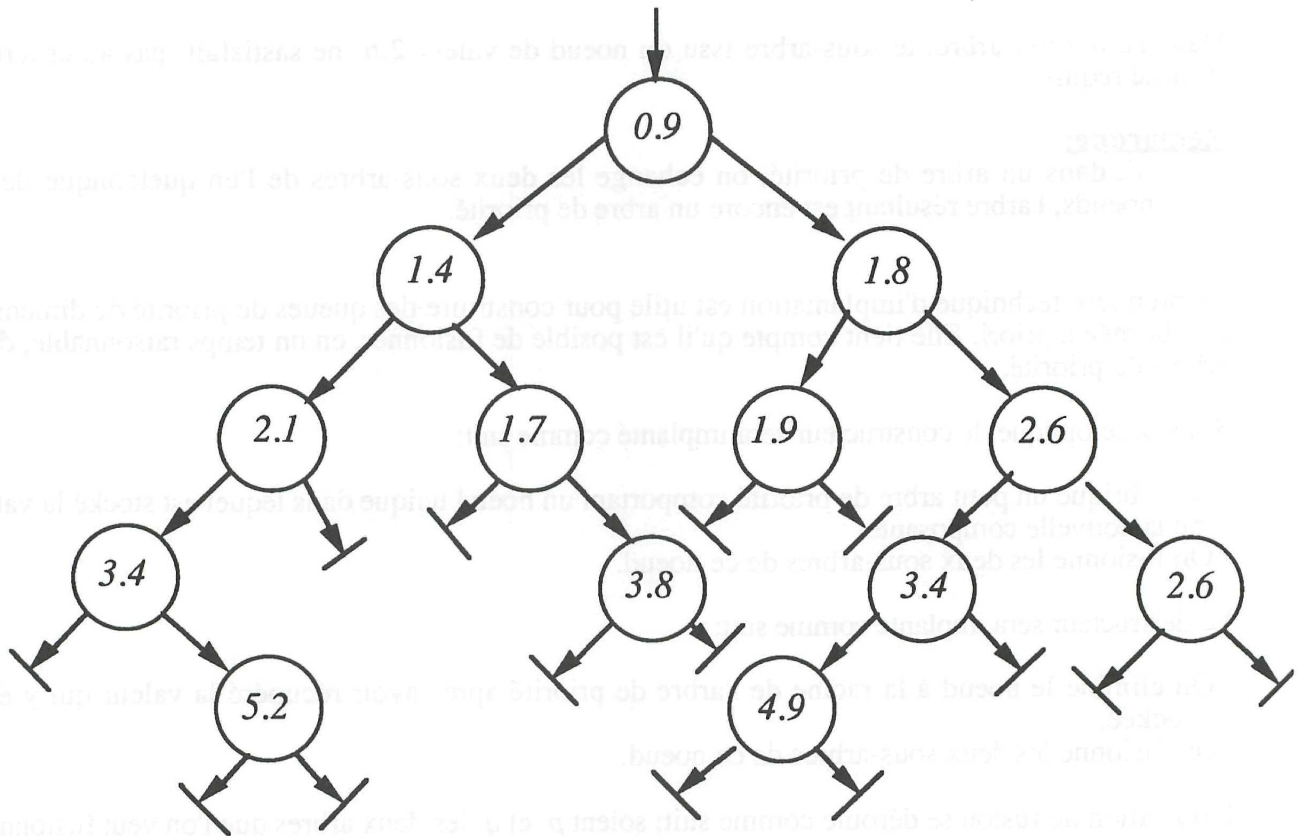


figure 38

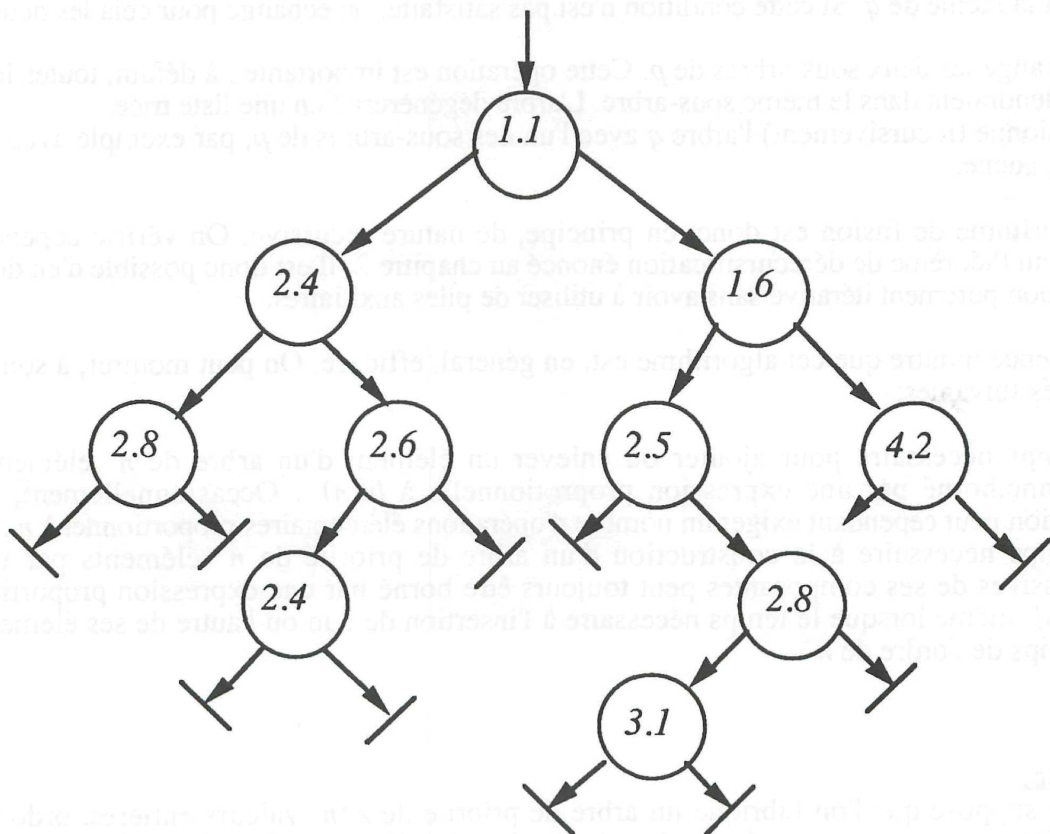


figure 39

Dans ce dernier arbre, le sous-arbre issu du noeud de valeur *2.6* ne sastisfait pas au critère de priorité requis.

Remarque:

Si, dans un arbre de priorité, on échange les deux sous-arbres de l'un quelconque de ses noeuds, l'arbre résultant est encore un arbre de priorité.

La première technique d'implantation est utile pour construire des queues de priorité de dimension non bornée à priori. Elle tient compte qu'il est posible de fusionner, en un temps raisonnable, deux arbres de priorité.

Sous cette optique, le constructeur sera implanté comme suit:

- On fabrique un petit arbre de priorité comportant un noeud unique dans lequel est stocké la valeur de la nouvelle composante.
- On fusionne les deux sous-arbres de ce noeud.

Le destructeur sera implanté comme suit:

- On élimine le noeud à la racine de l'arbre de priorité aprés avoir récupéré la valeur qui y était stockée.
- On fusionne les deux sous-arbres de ce noeud.

L'opération de fusion se déroule comme suit; soient *p* et *q* les deux arbres que l'on veut fusionner.

- Si *q* est vide, l'arbre fusionné est identique à *p*.
- Dans le cas général, on fait en sorte que le noeud à la racine de *p* soit prioritaire par rapport à celui situé à la racine de *q*. Si cette condition n'est pas satisfaite, on échange pour cela les deux arbres *p* et *q*.
- On échange les deux sous-arbres de *p*. Cette opération est importante : à défaut, toutes les fusions interviendraient dans le même sous-arbre. L'arbre dégénèrerait en une liste triée.
- On fusionne (récursivement) l'arbre *q* avec l'un des sous-arbres de *p*, par exemple avec son sous-arbre gauche.

Cet algorithme de fusion est donc, en principe, de nature récursive. On vérifie cependant qu'il satisfait au théorème de dérécursification énoncé au chapitre 2; il est donc possible d'en donner une formulation purement itérative sans avoir à utiliser de piles auxiliaires.

L'expérience montre que cet algorithme est, en général, efficace. On peut montrer, à son sujet, les propriétés suivantes:

- Le temps nécessaire pour ajouter ou enlever un élément d'un arbre de *n* éléments est en moyenne,borné par une expression proprotionnelle à *ln(n)* . Occasionnellement, une telle opération peut cependant exiger un nombre d'opérations élémentaires proportionnel à *n* .
- Le temps nécessaire à la construction d'un arbre de priorité de *n* éléments par insertions successives de ses composantes peut toujours être borné par une expression proportionnelle à *n*ln(n)*, même lorsque le temps nécessaire à l'insertion de l'un ou l'autre de ses éléments a pris un temps de l'ordre de *n* .

Exemple:

On suppose que l'on fabrique un arbre de priorité de *2*n* valeurs entières, ordonnées par ordre croissant, en y insérant, dans cet ordre, des éléments de valeurs *2*n-1*, *2*n*, *2*n-3*, *2*n-2*, *2*n-5*, *2*n-4*,...,*1* et *2* . On vérifie que cet arbre se construira très rapidement (en un temps de l'ordre de *n)*, toutes les insertions ayant lieu près de son sommet. Pour *n=5*, on aboutit à l'arbre de la figure 40.

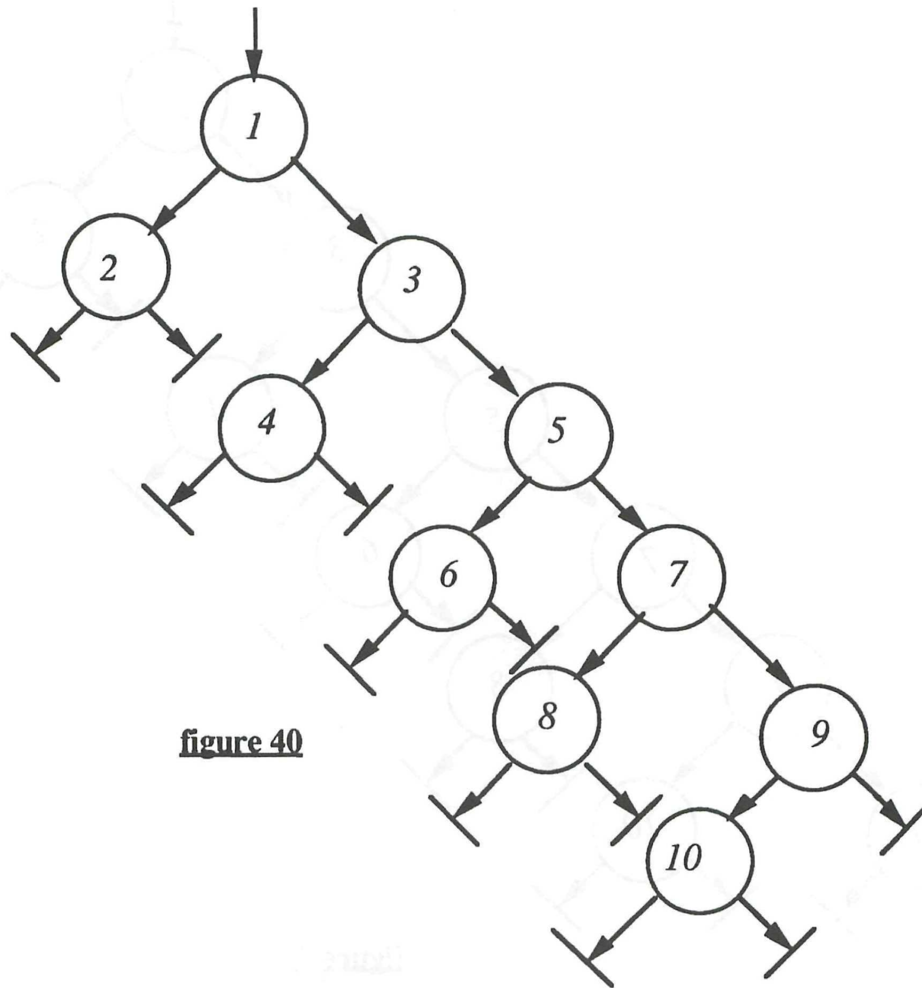


figure 40

Par contre si dans l'arbre résultant, on insère un nouvel élément de valeur *2*n+1*, on vérifie que l'arbre va basculer. Primitivement orienté à droite, il va se déporter sur la gauche. C'est dans un tel cas qu'une insertion prend un temps de l'ordre de n . Ainsi, on vérifie que l'insertion d'un nouvel élément de valeur *11* dans l'arbre de la figure 40 résulte dans celui de la figure 41.

$\bar{t}(n)$ = tps construct° moyen arbre n élts

$t(n,j)$ = " " " , hyp. j^e élt est prioritaire

insert° du j^e élt

$$t(n,j) = t(j-1) + \alpha + (n-j)\beta + t((n-j)-(n-j)\%2) + t(j-1+(n-j)\%2) - t(j-1)$$

$$t(n) = \frac{1}{n}\sum_{j=1}^{n} t(n,j) = \alpha + \frac{\beta}{n}\sum_{j=1}^{n}(n-j) + \frac{1}{n}\sum_{j=1}^{n} t((n-j)-(n-j)\%2) + \frac{1}{n}\sum_{j=1}^{n} t(j-1+(n-j)\%2)$$

$n-j=k$
$j=n-k$

$$= \alpha + \frac{\beta}{n}\underbrace{\sum_{k=0}^{n-1} k}_{\frac{n(n-1)}{2}} + \frac{1}{n}\sum_{k=0}^{n-1} t(k-k\%2) + \frac{1}{n}\sum_{k=0}^{n-1} t(n-k-1+k\%2)$$

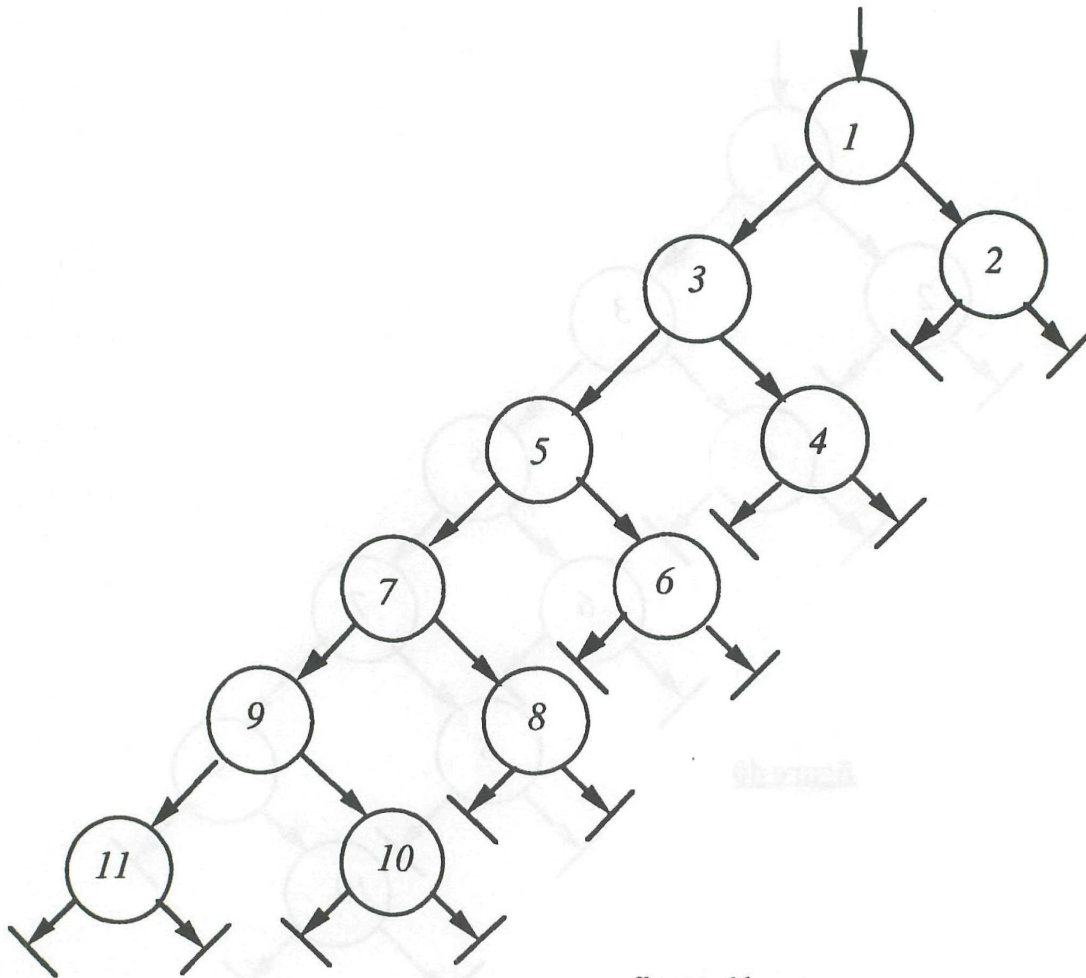


figure 41

Par contre si on continue à insérer des éléments dans l'arbre résultant de ce basculement, on constate que quelles que soient leurs valeurs, il n'y aura pas (dans l'immédiat) de nouveau basculement.Les insertions seront rapides: on vérifie qu'elles interviennent en priorité dans les sous-arbre de faible profondeur. On vérifierait ainsi que l'on n'a jamais une succession d'opérations d'insertion qui exigent toutes un temps de l'ordre de n ; une opération coûteuse sera isolée: ella pour effet de restructurer l'arbre de manière à rendre très rapides les opérations suivantes:

Une queue de priorité de capacité bornée à priori peut-être implantée au moyen d'une rangée et d'un compteur auxiliaire de valeur égale au nombre d'éléments de la queue. L'idée est destocker dans la rangée un arbre de priorité à liaisons implicites. Cet arbre sera équilibré dans le sens suivant:

- Tous les niveaux de l'arbre, sauf celui de profondeur maximum sont pleins.
- Le niveau de profondeur maximum est rempli, de proche en proche, de gauche à droite.

La figure 42 montre un tel arbre de priorité équilibré de douze éléments.

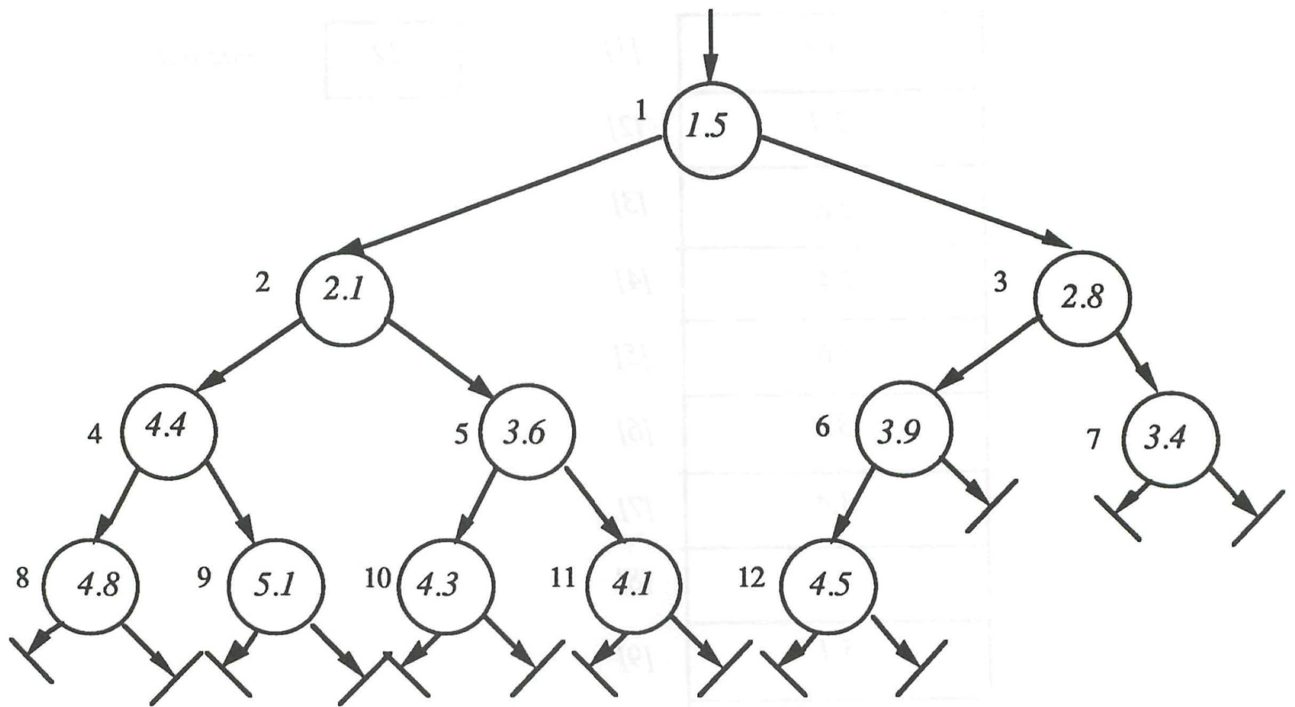


figure 42

Un tel arbre sera stocké dans la rangée représentative étage par étage; les éléments d'un même niveau seront pris de gauche à droite. Ainsi, l'arbre de la figure 42 aura la représentation suivante dans le cas d'une rangée d'une capacité de quinze éléments (figure 43).

$$= \alpha + \beta \frac{n-1}{2} + \frac{1}{n}\left[t(0) + t(1) + t(1) + t(2) + t(2) + \ldots + t\left((n-1) - (n-1)\,\%\,2\right) \right]$$

$$= \frac{1}{2}\left[t(n-1) + t(n-2) + t(n-2) + t(n-3) + t(n-3) + \ldots + t\left((n-1)\,\%\,2\right) \right]$$

$$t(n) = \alpha + \beta \frac{n-1}{2} + \frac{1}{n} t(0) + \frac{2}{n} \sum_{k=1}^{n-2} t(k) + \frac{1}{n} t(n-1) + \frac{1}{n} t\left((n-1) - (n-1) \text{ div } 2\right)$$

$$\left(+ \frac{1}{n} t\left((n-1) \text{ div } 2\right) \right) \leq \frac{1}{n} t(n-1)$$

$$t(n) \leq \alpha + \beta \frac{n-1}{2} + \frac{2}{n} \sum_{k=1}^{n-1} t(k)$$

$$\left\{ u(n) \leq \alpha + \beta \frac{n-1}{2} + \frac{2}{n} \sum_{k=1}^{n-1} t(k) \text{ et } u(0) = 0 \; ; \; t(j) \leq u(j) \right.$$

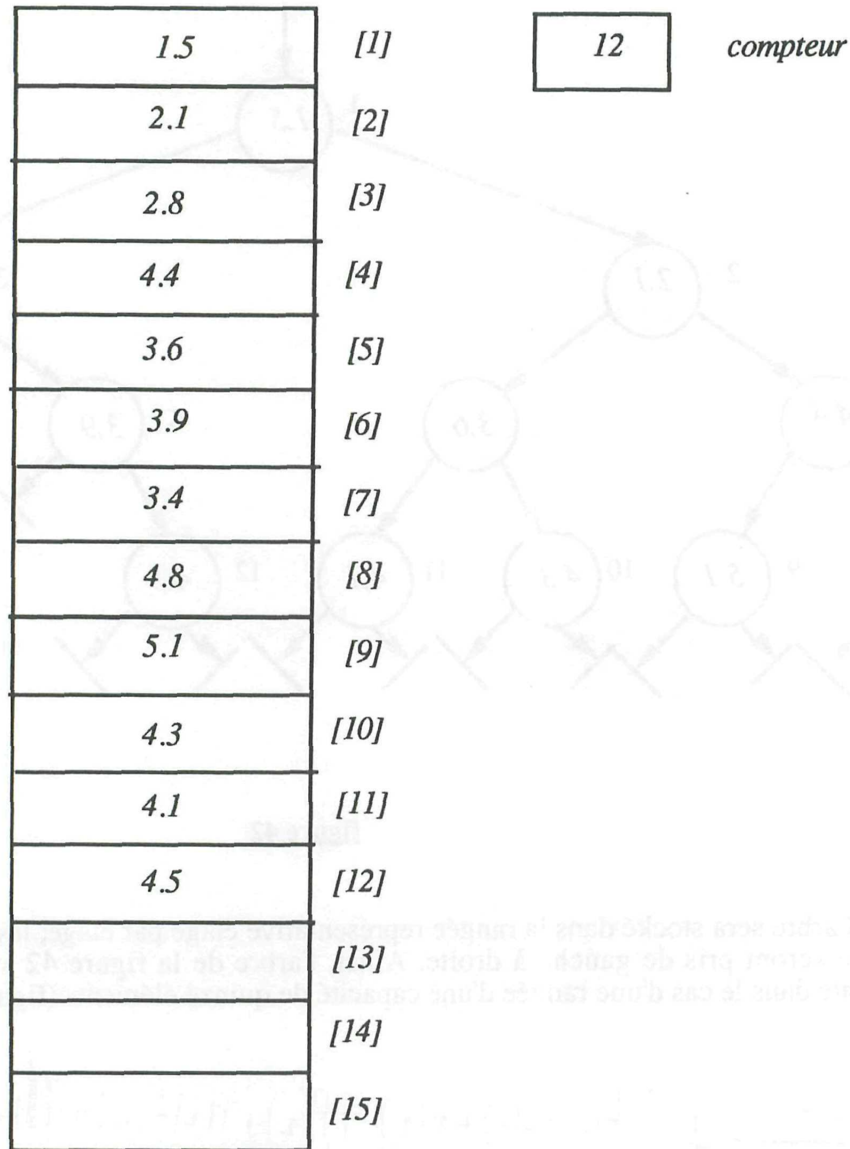


Fig. 43

Dans cette représentation implicite d'un arbre de priorité, on vérifie qu'il est facile de monter ou de descendre dans l'arbre. Ainsi les deux fils de l'élément stocké en *element[k]* sont situés (s'il existent) en *élément[2*k]* et *élément[2*k+1]* . Le père de l'élément stocké en *élément[k]* se trouve en *élément[k % 2]* (pour autant, bien sûr, que *k>1*).

Ceci permet de réaliser le constructeur et le destructeur de manière telle que le temps nécessaire à l'insertion d'un nouvel élément et à l'élimination de l'élément prioritaire d'un arbre de *n* composantes est dans tous les cas bornée par une expression proportionnelle à *ln(n)*.

Pour insérer un nouvel élément dans l'arbre, on considère le chemin issu de la racine et aboutissant à l'emplacement qu'il s'agit de remplir. L'élément nouveau est introduit à l'endroit approprié de ce chemin. Pour celà, les éléments par rapports auxquels l'élément nouveau est prioritaire sont tout d'abord déplacés vers le bas: ceci est facilement réalisable au moyen d'un parcours de bas en haut du chemin. La figure 44 montre le résultat de l'insertion d'un élément de valeur *2.6* dans l'arbre de la figure 42.

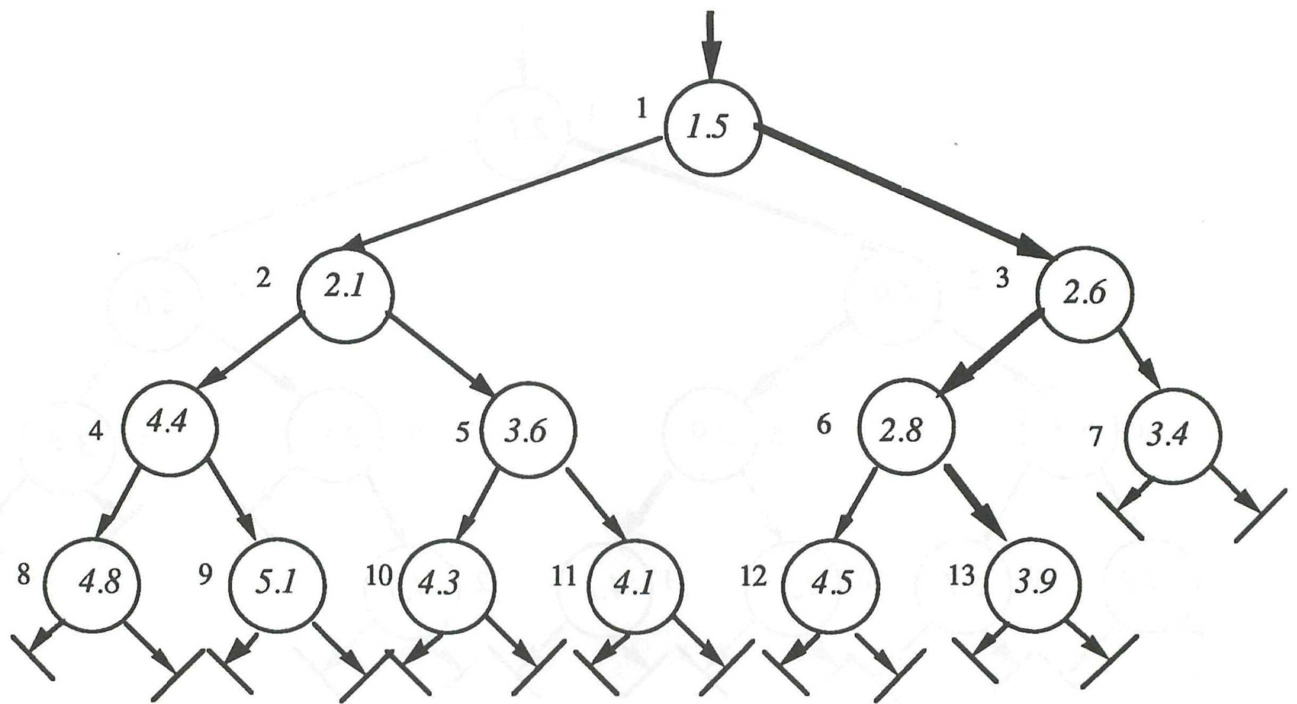


figure 44

L'élément prioritaire de l'arbre se trouve évidemment à sa racine. Lorsqu'on l'élimine, il est nécessaire de réinsérer l'élément le plus à droite du niveau inférieur de manière à rétablir un arbre de priorité équilibré. Il suffit pour celà de réinsérer l'élément concerné à l'endroit approprié du chemin issu de la racine et passant par le fils prioritaire de chaque noeud traversé. Les éléments de ce chemin qui sont prioritaires par rapport à celui que l'on réinsère sont tout d'abord déplacés vers le haut: ceci est facilement réalisable au moyen d'un parcours de haut en bas du chemin. La figure 45 montre le résultat de l'élimination de l'élément prioritaire de l'arbre de la figure 44.

$$n u(n) = n\alpha + \beta\frac{n(n-1)}{2} + 2\sum_{k=1}^{n-1} u(k) \quad \Big| \; (-1)$$

$$(n+1)u(n+1) = (n+1)\alpha + \beta\frac{(n+1)n}{2} + 2\sum_{k=1}^{n} u(k) \quad \Big| \; 1$$

$$(n+1)u(n+1) - n u(n) = \alpha + n\beta + 2u(n)$$

$$(n+1)u(n+1) - (n+2)u(n) = \alpha + n\beta$$

$$(n+1)\underbrace{[u(n+1) - u(n)]}_{\delta(n)} - u(n) = \alpha + n\beta$$

$$(n+1)\delta(n) - u(n) = \alpha + n\beta \quad \Big| \; (-1)$$

$$(n+2)\delta(n+1) - u(n+1) = \alpha + (n+1)\beta \quad \Big| \; 1$$

$$(n+2)\delta(n+1) - (n+2)\delta(n) = \beta$$

$$\delta(n+1) = \delta(n) + \frac{\beta}{n+2}$$

$$\delta(n) = \delta(1) + \beta\left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1}\right)$$

$$H(n) = \sum_{j=1}^{n} \frac{1}{j}$$

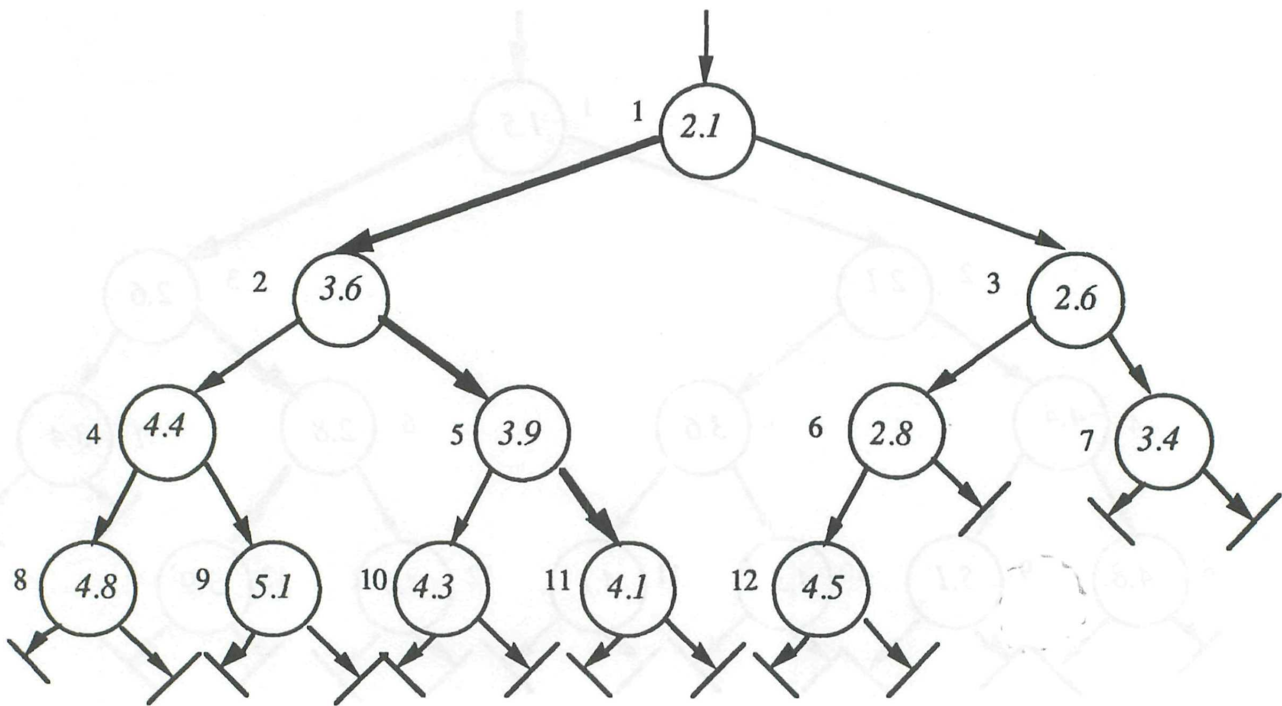


figure 45

Une queue de priorité est utilisable comme une boîte de tri dans le sens que si l'on introduit une suite d'éléments dans un ordre quelconque, puis qu'on les retire tous de la queue, ces éléments en ressortiront triés selon le critère de priorité incorporé dans la queue lors de sa construction. Ceci est illustré dans le programme *boîtes_tris* dont les déclarations principales sont données à la figure 46.

Ce programme ne nécessite que peu de commentaires. On y reconnaît tout d'abord le module *classage* et la classe protocole *queue_priorité* qui y est incorporée.

$\delta(n)=\delta(1)+\beta(H(n+1)-1) \underset{n\to\infty}{=} O(\ln n)$

$u(n+1)=u(n)+\delta(1)+\beta(H(n+1)-1)$

$u(n)=n\delta(1)-n\beta+\beta\sum_{j=1}^{n}H(j) \underset{n\to\infty}{=} O(n\ln n)$

program *boîtes_tris*

integer **subrange** *naturel*

naturel **subrange** *positif*

module *classage*

Boolean **functor** *relation*

actor *constructeur*

real **functor** *selecteur*

actor *destructeur*

class *queue_priorité*

queue_priorité **function** *queue_priorité_arborescente*

procedure *transvaser*

module *classage_borné*

Boolean **functor** *interrogateur*

class *queue_priorité_bornée*

queue_priorité_bornée function queue_priorité_rangée

constant *limite, écart*

relation **value** *inf, sub, absinf, classent*

queue_priorité_bornée **value** *boîte_tri_classent,*
boîte_tri_croissant

queue_priorité **value** *boîte_tri_décroissant,*
boîte_tri_absolu

Fig. 46

Ce module est suivi de la fonction génératrice *queue_priorité_arborescente* . Celle-ci fournit une implantation d'une queue de priorité non bornée à priori au moyen d'un arbre de priorité.

```
boites_tris                                              Vax Newton Compiler 0.2c16
                                                Page    1
                   Source listing


/* /*OLDSOURCE=BOITES_TRIS.NEW*/ */
PROGRAM boites_tris DECLARE

  integer SUBRANGE naturel(naturel>=0)
          SUBRANGE positif(positif>0);

MODULE classage
ATTRIBUTE queue_priorite,relation,constructeur,selecteur,destructeur
DECLARE(*classage*)
  Boolean FUNCTOR relation(real VALUE gauche,droite);
  ACTOR constructeur(real VALUE val);
  real FUNCTOR selecteur;
  ACTOR destructeur(real REFERENCE var);

  CLASS queue_priorite
    VALUE moi
    ATTRIBUTE prioritaire,vide,longueur,introduire,premier,retirer
    (relation VALUE prioritaire;
     constructeur VALUE cons;
     selecteur VALUE prem;
     destructeur VALUE des)
  (*Un objet de la classe  queue_priorite  est une queue de valeurs
   relles ordonnees selon la relation  prioritaire ; cette derniere
  satisfera aux contraintes suivantes pour tout ensemble de valeurs
    reelles  x ,  y  et  z :

      prioritaire[x,y] NAND prioritaire[y,x]

      prioritaire[x,y] NOR prioritaire[y,x] |>
        (prioritaire[z,x]==prioritaire[z,y])

      prioritaire[x,y]/\prioritaire[y,z]|>prioritaire[x,z]

    L'implanteur fournira les objets proceduraux suivants:

    cons[val]    Insere dans la queue un nouvel element de valeurval
    prem EVAL    Livre la valeur de l'element prioritaire
   des[var]    Elimine de la queue son element prioritaire apres en
                   avoir stocke la valeur dans la variable  var
  *)
  DECLARE(*classage*)
    naturel VARIABLE compte VALUE longueur:=0;
    (*Le nombre d'elements de la queue concernee*)

    Boolean EXPRESSION vide=
    (*Vrai ssi la queue est vide*)
      longueur=0;

    queue_priorite FUNCTION introduire
      (real VALUE val)
    (*Place dans la queue un nouvel element de valeur  val .
      Le resultat est la queue concernee.
```

```
      *)
    DO compte:=SUCC compte; cons[val] TAKE moi DONE;

    real EXPRESSION premier=
    (*La valeur de l'element prioritaire.

      Condition d'emploi: ~vide
    *)
     (IF longueur=0 THEN
      print("###premier porte sur queue de priorite vide###",line)
      RETURN DONE;
     prem EVAL);

  queue_priorite FUNCTION retirer
   (real REFERENCE var)
  (*Retire de la queue son element prioritaire et en place la valeur
   dans la variable designee par  var . Le resultat est la queue
   concernee.

      Condition d'emploi: ~vide
    *)
    DO(*retirer*)
      IF longueur=0 THEN
       print("###retirer porte sur queue de priorite vide###",line)
      RETURN DONE;
      des[var]; compte:=PRED compte
    TAKE moi DONE(*retirer*)
  DO(*queue_priorite*)DONE
DO(*classage*)DONE;

  queue_priorite FUNCTION queue_priorite_arborescente
    (relation VALUE prior)
  (*Cette fonction generatrice livre une queue de priorite, non
    bornee a priori, de valeurs rellees ordonnee selon la relation
    d'ordre  prior .
  *)
  DECLARE(*queue_priorite_arborescente*)
   OBJECT arbre_priorite
     (real VALUE membre; arbre_priorite VARIABLE gauche,droite)
     VARIABLE racine:=NIL;

   arbre_priorite FUNCTION fusion
     (arbre_priorite VARIABLE p,q)
   (*La fusion des deux arbres donnes  p  et  q *)
   DECLARE arbre_priorite REFERENCE r->p DO
    WITHIN q REPEAT
      IF TAKE
        UNLESS r=NIL THEN
          prior[(*q.*)membre,r.membre]
        DEFAULT TRUE DONE
      THEN r:=:q DONE;
      (*Maintenant,  r.membre  est l'element prioritaire*
```

```
boites_tris                                    Vax Newton Compiler 0.2c16
                              Page      3
                Source listing

            CONNECT r THEN
              r->gauche:=:droite
            DONE
          REPETITION
        TAKE p DONE
      DO(*queue_priorite_arborescente*)TAKE
        queue_priorite
          (prior,
           BODY
             constructeur(real VALUE val)
           DO
             racine:=fusion(racine,arbre_priorite(val,NIL,NIL))
           DONE,
           BODY selecteur DO TAKE racine.membre DONE,
           BODY
             destructeur(real REFERENCE var)
           DO
             CONNECT racine THEN
               var:=membre; racine:=fusion(gauche,droite)
             DONE
           DONE)
      DONE(*queue_priorite_arborescente*);

      PROCEDURE transvaser
        (queue_priorite VALUE source,destination)
      (*Transfere l'ensemble des elements de la queue de priorite
        source  dans la queue de priorite  destination
      *)
      DECLARE real VARIABLE x DO
        UNTIL source.vide REPEAT
          source.retirer(x);
          edit(x,15,8);
          IF source.longueur\5=0 THEN line DONE;
          destination.introduire(x)
        REPETITION;
        print("<<<FIN>>>",page)
      DONE(*transvaser*);
    /* /*EJECT*/ */
```

La procédure *transvaser* a pour effet de faire passer tous les éléments de la queue de priorité *source* dans la queue de priorité *destination* en imprimant leurs valeurs au passage. Les éléments seront donc imprimés dans l'ordre impliqué par le critère de priorité dans la queue *source*.

```
MODULE classage_borne
  ATTRIBUTE queue_priorite_bornee,interrogateur
DECLARE(*classage_borne*)
  Boolean FUNCTOR interrogateur;

  CLASS queue_priorite_bornee
    VALUE moi
    ATTRIBUTE capacite,prioritaire,vide,plein,longueur,
              introduire,premier,retirer,
              queue_priorite_induite
     (positif VALUE capacite; relation VALUE prioritaire;
      interrogateur VALUE vd,pln;
      constructeur VALUE cons;
      selecteur VALUE sel;
      destructeur VALUE des)
   (*Un objet de la classe  queue_priorite_bornee  est une queue
    d'au maximum  capacite  valeurs reelles, ordonnees selon la
    relation  prioritaire.  L'implanteur fournira les objets
    proceduraux suivants:

    prioritaire[x,y]  vrai ssi  x  est prioritaire par rapport a  y
      vd EVAL         vrai ssi la queue est vide
      pln EVAL        vrai ssi la queue est pleine
      cons[val]       vrai insere dans la queue un nouvel element de
                      valeur  val
      sel EVAL        l'element prioritaire de la queue
      des[var]        elimine de la queue son element prioritaire et
                      en stocke la valeur dans la variable  var

    Condition d'emploi:

      prioritaire[x,y] NAND prioritaire[y,x]
      prioritaire[x,y] NOR prioritaire[y,x] |>
        (prioritaire[x,z]==prioritaire[y,z])
      prioritaire[x,y]/\prioritaire[y,z] |> prioritaire[x,z]
   *)
   DECLARE(*queue_priorite_bornee*)
    queue_priorite VALUE queue_priorite_induite=
     (*La queue concernee vue comme queue de priorite generale*)
      queue_priorite(prioritaire,cons,sel,des);

    Boolean EXPRESSION
      vide=vd EVAL(*Vrai ssi la queue est vide*),
      plein=pln EVAL(*Vrai ssi la queue est pleine*);

    naturel EXPRESSION longueur=
     (*Le nombre d'elements de la queue concernee*)
      queue_priorite_induite.longueur;

    queue_priorite_bornee FUNCTION introduire
      (real VALUE x)
     (*Insere dans la queue concernee un nouvel element de valeur  x
```

```
      Le resultat est la queue concernee.

      Condition d'emploi: ~plein
    *)
    DO(*introduire*)
      IF pln EVAL THEN
        print("###introduire porte sur queue priorite pleine###",
           line)
      RETURN DONE;
      queue_priorite_induite.introduire(x)
    TAKE moi DONE(*introduire*);

    real EXPRESSION premier=
    (*L'element prioritaire de la queue.

      Condition d'emploi: ~vide
    *)
      queue_priorite_induite.premier;

    queue_priorite_bornee FUNCTION retirer
      (real REFERENCE xx)
    (*Elimine de la queue son element prioritaire et en stocke la
      valeur dans la variable designee par  xx ; le resultat est
      la queue de priorite concernee.

      Condition d'emploi: ~vide
    *)
    DO queue_priorite_induite.retirer(xx) TAKE moi DONE
  DO(*queue_priorite_bornee*)DONE
DO(*classage_borne*)DONE;

  queue_priorite_bornee FUNCTION queue_priorite_rangee
    (positif VALUE capacite; relation VALUE prior)
  (*Etablit une queue de priorite d'au maximum  capacite  valeurs
    reelles ordonnees selon la relation d'ordre  prior
  *)
  DECLARE(*queue_priorite_rangee*)
    integer VARIABLE compte:=0;

    real ROW vecteur VALUE tab=vecteur(1 TO capacite)
  DO(*queue_priorite_rangee*)TAKE
    queue_priorite_bornee
      (capacite,prior,
       BODY interrogateur DO TAKE compte=0 DONE,
       BODY interrogateur DO TAKE compte=capacite DONE,
       BODY
         constructeur(real VALUE val)
       DECLARE
         integer VARIABLE fils:=(compte:=SUCC compte),pere
       DO
         WHILE TAKE
          IF fils>1 THEN
```

```
boites_tris                                          Vax Newton Compiler 0.2c16
                                              Page    6
                          Source listing

                 prior[val,tab[(pere:=fils%2)]]
               DEFAULT FALSE DONE
             REPEAT tab[fils]:=tab[(fils:=pere)] REPETITION;
             tab[fils]:=val
           DONE(*constructeur*),
           BODY selecteur DO TAKE tab[1] DONE,
           BODY
             destructeur(real REFERENCE var)
           DECLARE
             integer VARIABLE pere,aine,puine,fils;
             real VALUE val=tab[compte]
           DO
             var:=tab[1];
             IF (compte:=PRED compte)>0 THEN
               pere:=1;
               CYCLE descente REPEAT

               IF
                 (aine:=2*pere)>compte
               EXIT descente DONE;

                 fils:=
                  IF aine=compte THEN aine ELSE
                  IF prior[tab[aine],tab[(puine:=SUCC aine)]] THEN aine
                  DEFAULT puine DONE;

               UNLESS
                 prior[tab[fils],val]
               EXIT descente DONE;

                 tab[pere]:=tab[(pere:=fils)]
               REPETITION;
               tab[pere]:=val
             DONE(*compte>1*)
           DONE(*destructeur*))
      DONE(*queue_priorite_rangee*);
 /* /*EJECT*/ */
```

Le module *classage_borné* inclut la classe protocole *queue_priorité_bornée* ; celle-ci décrit l'interface d'une queue de priorité de capacité bornée à priori. La fonction génératrice subséquente *queue_priorité_rangée* implante une queue de priorité de capacité bornée à priori au moyen d'une rangée selon la technique décrite plus haut (arbre de priorité implicite). On remarque que la programmation du constructeur et surtout du destructeur exige quelques précautions.

```
boites_tris                                    Vax Newton Compiler 0.2c16
                                 Page      7
                Source listing

   CONSTANT limite=200,ecart=10;

   relation VALUE
      inf=BODY relation DO TAKE gauche<droite DONE,
      sup=BODY relation DO TAKE gauche>droite DONE,
      absinf=BODY relation DO TAKE ABS gauche<ABS droite DONE,
      classent=BODY relation DO TAKE FLOOR gauche<FLOOR droite DONE;

   queue_priorite_bornee VALUE
      boite_tri_classent=queue_priorite_rangee(limite,classent),
      boite_tri_croissant=queue_priorite_rangee(limite,inf);
   queue_priorite VALUE
      boite_tri_decroissant=queue_priorite_arborescente(sup),
      boite_tri_absolu=queue_priorite_arborescente(absinf);
   real VARIABLE x
DO(*boites_tris*)
   print("*****Construction d'un echantillon*****",line);
   UNTIL boite_tri_classent.plein REPEAT
      boite_tri_classent.introduire((x:=ecart*normal));
      edit(x,15,8);
      IF boite_tri_classent.longueur\5=0 THEN line DONE
   REPETITION;
   print("<<<FIN>>>",page);
   print("***Echantillon vu par classes entieres croissantes***",
         line);
   transvaser(boite_tri_classent.queue_priorite_induite,
              boite_tri_decroissant);
   print("***Echantillon vu par ordre decroissant***",line);
   transvaser(boite_tri_decroissant,boite_tri_absolu);
   print("***Echantillon vu par ordre absolu croissant***",
         line);
   transvaser(boite_tri_absolu,
              boite_tri_croissant.queue_priorite_induite);
   print("***Echantillon vu par ordre croissant***",line);
   UNTIL boite_tri_croissant.vide REPEAT
      boite_tri_croissant.retirer(x);
      edit(x,15,8);
      IF boite_tri_croissant.longueur\5=0 THEN line DONE
   REPETITION;
   print("<<<<<FIN>>>>>")
DONE(*boites_tris*)
```

**** No messages were issued ****

Ce programme va utiliser quatre queues de priorités; ces dernières fonctionneront toutes comme boites de tri.

```
*****Construction d'un echantillon*****
    3.11200339    6.75430833   18.54519386    8.70235076    2.98500725
  -12.12998379  -14.84830470   -8.31167025    5.51073477   -5.91503132
   18.02794447   -4.37713352  -17.14467035    4.96646836    2.52895943
    5.24352159   -5.71359602   -8.27074182   -1.51651527   12.50355096
    3.08246870   -8.50888284   -3.91193490    8.20978382   -1.06194261
   22.68388520   -2.06067266   -1.34735748    7.52512352  -10.52718951
    2.32278228   -2.10084739   -3.61998630    -.91103659    -.45114404
    6.71197312   21.45119500  -11.45147589    1.61432553   -2.18711432
   10.07298108   -7.93059616   -6.90087122   10.00444540    4.01962606
   11.24194564    9.74893859   -8.73197068    1.96126931  -13.16301198
  -20.52021828   -8.44052148   -6.89676472   16.64963003  -15.16054654
   12.76356379    4.27607093  -17.30971475    6.11640310    4.22643310
    2.74777451   -7.25834184    -.71723736   11.21260762   13.05193918
    7.71088415   -4.78288433    4.04084789  -18.75734992     .47323090
   -2.38930333   -2.52155685    7.85460543   -2.69576566   14.87262723
   -3.73216365   -1.62206149    8.62360851   -8.86287054   -9.64521447
    7.04651224  -19.46032111    5.58953697   15.65402380    5.48238302
   -4.29400897   -3.98616773  -11.69397684     .13953984   -8.17116288
   -3.60593051   -1.42041743   -1.23465467   -6.83335346   -3.89058023
    5.71304394  -10.52355401  -11.98010909    4.72571873  -16.34011599
  -11.90548743   14.96246216   -3.15916844   11.04463138   22.17653924
   -7.58657528   -1.05425539     .06048231   16.06010567    -.10507990
    -.75139762   -8.86212592   11.76480826   -7.61357796    8.71496201
   -2.78933727    6.39741576    5.09234721    9.63181164   -8.75742422
  -14.76655916   -1.51226666   13.89971064   10.46384450   -4.25911260
   -1.05431724  -19.28262783    8.96830231   16.65242866    5.71930116
    8.54214938    -.45062413    7.35599621    3.64402286   16.69873762
   15.47603865    6.97045299   -2.84698943    8.43610508    3.26717165
   -3.16249661   -6.83498916   -9.44664980    -.12508947    8.85777763
  -23.11722184  -12.59212736    4.55878209    6.87681404    5.57673431
    8.88742944    7.85032549   -9.82497272    8.63149484   -6.61637182
    7.27778032    9.25952979   12.02142507    3.16440085   18.22375162
    8.44621143    7.45819907   -8.60167513   15.05692567     .26333193
   -5.77413565  -14.68792145   -5.62884370   12.07604714   -9.43006685
   -8.63164032   14.57536686    5.82942841    5.18197246    7.11623217
  -14.41818323   14.70087015   11.07407900  -12.92818039   -5.80017050
    -.34639462   20.14959376    3.21531168   -6.99608957   -7.66731069
    -.05280165   11.75425048    4.79198637    4.68049236    9.61159092
    9.11767113   11.40581906   17.44671782   -8.08861273    6.47937176
   -8.37893022   -8.18530187    9.95543845  -19.19134677   18.58665052
<<<FIN>>>
```

Le programme génère un échantillon de *limite = 200* valeurs aléatoires (distribution normale d'écart-type *écart = 10*). Cet échantillon est successivement placé dans chacune des quatre queues; il en ressort chaque fois trié selon le critère de priorité correspondant.

```
***Echantillon vu par classes entieres croissantes***
    -23.11722184   -20.52021828   -19.28262783   -19.19134677   -19.46032111
    -18.75734992   -17.14467035   -17.30971475   -16.34011599   -15.16054654
    -14.76655916   -14.84830470   -14.41818323   -14.68792145   -13.16301198
    -12.12998379   -12.92818039   -12.59212736   -11.90548743   -11.98010909
    -11.69397684   -11.45147589   -10.52718951   -10.52355401    -9.43006685
     -9.64521447    -9.82497272    -9.44664980    -8.75742422    -8.86212592
     -8.44052148    -8.73197068    -8.37893022    -8.08861273    -8.18530187
     -8.17116288    -8.63164032    -8.50888284    -8.60167513    -8.86287054
     -8.27074182    -8.31167025    -7.25834184    -7.61357796    -7.58657528
     -7.66731069    -7.93059616    -6.89676472    -6.83335346    -6.99608957
     -6.90087122    -6.61637182    -6.83498916    -5.91503132    -5.80017050
     -5.62884370    -5.77413565    -5.71359602    -4.25911260    -4.37713352
     -4.29400897    -4.78288433    -3.15916844    -3.91193490    -3.89058023
     -3.60593051    -3.98616773    -3.73216365    -3.16249661    -3.61998630
     -2.78933727    -2.06067266    -2.18711432    -2.69576566    -2.52155685
     -2.38930333    -2.84698943    -2.10084739    -1.05431724    -1.34735748
     -1.51226666    -1.05425539    -1.06194261    -1.23465467    -1.42041743
     -1.62206149    -1.51651527     -.71723736     -.75139762     -.10507990
      -.05280165     -.34639462     -.12508947     -.45114404     -.91103659
      -.45062413      .06048231      .13953984      .26333193      .47323090
      1.96126931     1.61432553     2.32278228     2.74777451     2.52895943
      2.98500725     3.21531168     3.26717165     3.08246870     3.16440085
      3.64402286     3.11200339     4.27607093     4.96646836     4.22643310
      4.01962606     4.72571873     4.04084789     4.79198637     4.68049236
      4.55878209     5.82942841     5.09234721     5.71304394     5.18197246
      5.48238302     5.58953697     5.57673431     5.51073477     5.24352159
      5.71930116     6.11640310     6.39741576     6.75430833     6.47937176
      6.71197312     6.97045299     6.87681404     7.11623217     7.85032549
      7.71088415     7.85460543     7.04651224     7.52512352     7.27778032
      7.45819907     7.35599621     8.62360851     8.71496201     8.54214938
      8.85777763     8.20978382     8.63149484     8.96830231     8.44621143
      8.88742944     8.43610508     8.70235076     9.63181164     9.61159092
      9.74893859     9.25952979     9.11767113     9.95543845    10.46384450
     10.00444540    10.07298108    11.07407900    11.40581906    11.76480826
     11.75425048    11.04463138    11.24194564    11.21260762    12.02142507
     12.76356379    12.50355096    12.07604714    13.89971064    13.05193918
     14.87262723    14.70087015    14.96246216    14.57536686    15.47603865
     15.65402380    15.05692567    16.06010567    16.64963003    16.69873762
     16.65242866    17.44671782    18.02794447    18.58665052    18.54519386
     18.22375162    20.14959376    21.45119500    22.68388520    22.17653924
<<<FIN>>>
```

Tout d'abord, le passage dans le queue de priorité bornée *boite_tri_classent* le fera trier par classes entières croissantes, c'est-à-dire que les valeurs de même partie entière seront considérées comme équivalentes: elles sortent de la queue dans un ordre arbitraire.

```
***Echantillon vu par ordre decroissant***
    22.68388520    22.17653924    21.45119500    20.14959376    18.58665052
    18.54519386    18.22375162    18.02794447    17.44671782    16.69873762
    16.65242866    16.64963003    16.06010567    15.65402380    15.47603865
    15.05692567    14.96246216    14.87262723    14.70087015    14.57536686
    13.89971064    13.05193918    12.76356379    12.50355096    12.07604714
    12.02142507    11.76480826    11.75425048    11.40581906    11.24194564
    11.21260762    11.07407900    11.04463138    10.46384450    10.07298108
    10.00444540     9.95543845     9.74893859     9.63181164     9.61159092
     9.25952979     9.11767113     8.96830231     8.88742944     8.85777763
     8.71496201     8.70235076     8.63149484     8.62360851     8.54214938
     8.44621143     8.43610508     8.20978382     7.85460543     7.85032549
     7.71088415     7.52512352     7.45819907     7.35599621     7.27778032
     7.11623217     7.04651224     6.97045299     6.87681404     6.75430833
     6.71197312     6.47937176     6.39741576     6.11640310     5.82942841
     5.71930116     5.71304394     5.58953697     5.57673431     5.51073477
     5.48238302     5.24352159     5.18197246     5.09234721     4.96646836
     4.79198637     4.72571873     4.68049236     4.55878209     4.27607093
     4.22643310     4.04084789     4.01962606     3.64402286     3.26717165
     3.21531168     3.16440085     3.11200339     3.08246870     2.98500725
     2.74777451     2.52895943     2.32278228     1.96126931     1.61432553
      .47323090      .26333193      .13953984      .06048231     -.05280165
     -.10507990     -.12508947     -.34639462     -.45062413     -.45114404
     -.71723736     -.75139762     -.91103659    -1.05425539    -1.05431724
    -1.06194261    -1.23465467    -1.34735748    -1.42041743    -1.51226666
    -1.51651527    -1.62206149    -2.06067266    -2.10084739    -2.18711432
    -2.38930333    -2.52155685    -2.69576566    -2.78933727    -2.84698943
    -3.15916844    -3.16249661    -3.60593051    -3.61998630    -3.73216365
    -3.89058023    -3.91193490    -3.98616773    -4.25911260    -4.29400897
    -4.37713352    -4.78288433    -5.62884370    -5.71359602    -5.77413565
    -5.80017050    -5.91503132    -6.61637182    -6.83335346    -6.83498916
    -6.89676472    -6.90087122    -6.99608957    -7.25834184    -7.58657528
    -7.61357796    -7.66731069    -7.93059616    -8.08861273    -8.17116288
    -8.18530187    -8.27074182    -8.31167025    -8.37893022    -8.44052148
    -8.50888284    -8.60167513    -8.63164032    -8.73197068    -8.75742422
    -8.86212592    -8.86287054    -9.43006685    -9.44664980    -9.64521447
    -9.82497272   -10.52355401   -10.52718951   -11.45147589   -11.69397684
   -11.90548743   -11.98010909   -12.12998379   -12.59212736   -12.92818039
   -13.16301198   -14.41818323   -14.68792145   -14.76655916   -14.84830470
   -15.16054654   -16.34011599   -17.14467035   -17.30971475   -18.75734992
   -19.19134677   -19.28262783   -19.46032111   -20.52021828   -23.11722184
<<<FIN>>>
```

De cette queue, l'echantillon est transvasé dans la queue de priorité non bornée *boite_tri_croissant* . Il en sortira trié dans l'ordre décroissant.
L'échantillon est ensuite transvasé de cette queue dans le queue de priorité non bornée *boite_tri_absolu*. Cette dernière implique un tri dans l'ordre des valeurs absolues croissantes.

```
***Echantillon vu par ordre absolu croissant***
   -.05280165    .06048231   -.10507990   -.12508947    .13953984
    .26333193   -.34639462   -.45062413   -.45114404    .47323090
   -.71723736   -.75139762   -.91103659  -1.05425539  -1.05431724
  -1.06194261  -1.23465467  -1.34735748  -1.42041743  -1.51226666
  -1.51651527   1.61432553  -1.62206149   1.96126931  -2.06067266
  -2.10084739  -2.18711432   2.32278228  -2.38930333  -2.52155685
   2.52895943  -2.69576566   2.74777451  -2.78933727  -2.84698943
   2.98500725   3.08246870   3.11200339  -3.15916844  -3.16249661
   3.16440085   3.21531168   3.26717165  -3.60593051  -3.61998630
   3.64402286  -3.73216365  -3.89058023  -3.91193490  -3.98616773
   4.01962606   4.04084789   4.22643310  -4.25911260   4.27607093
  -4.29400897  -4.37713352   4.55878209   4.68049236   4.72571873
  -4.78288433   4.79198637   4.96646836   5.09234721   5.18197246
   5.24352159   5.48238302   5.51073477   5.57673431   5.58953697
  -5.62884370   5.71304394  -5.71359602   5.71930116  -5.77413565
  -5.80017050   5.82942841  -5.91503132   6.11640310   6.39741576
   6.47937176  -6.61637182   6.71197312   6.75430833  -6.83335346
  -6.83498916   6.87681404  -6.89676472  -6.90087122   6.97045299
  -6.99608957   7.04651224   7.11623217  -7.25834184   7.27778032
   7.35599621   7.45819907   7.52512352  -7.58657528  -7.61357796
  -7.66731069   7.71088415   7.85032549   7.85460543  -7.93059616
  -8.08861273  -8.17116288  -8.18530187   8.20978382  -8.27074182
  -8.31167025  -8.37893022   8.43610508  -8.44052148   8.44621143
  -8.50888284   8.54214938  -8.60167513   8.62360851   8.63149484
  -8.63164032   8.70235076   8.71496201  -8.73197068  -8.75742422
   8.85777763  -8.86212592  -8.86287054   8.88742944   8.96830231
   9.11767113   9.25952979  -9.43006685  -9.44664980   9.61159092
   9.63181164  -9.64521447   9.74893859  -9.82497272   9.95543845
  10.00444540  10.07298108  10.46384450 -10.52355401 -10.52718951
  11.04463138  11.07407900  11.21260762  11.24194564  11.40581906
 -11.45147589 -11.69397684  11.75425048  11.76480826 -11.90548743
 -11.98010909  12.02142507  12.07604714 -12.12998379  12.50355096
 -12.59212736  12.76356379 -12.92818039  13.05193918 -13.16301198
  13.89971064 -14.41818323  14.57536686 -14.68792145  14.70087015
 -14.76655916 -14.84830470  14.87262723  14.96246216  15.05692567
 -15.16054654  15.47603865  15.65402380  16.06010567 -16.34011599
  16.64963003  16.65242866  16.69873762 -17.14467035 -17.30971475
  17.44671782  18.02794447  18.22375162  18.54519386  18.58665052
 -18.75734992 -19.19134677 -19.28262783 -19.46032111  20.14959376
 -20.52021828  21.45119500  22.17653924  22.68388520 -23.11722184
<<<FIN>>>
```

Un dernier transvasage dans la queue de priorité bornée *boite_tri_croissant* intervient ensuite. La destruction de cette queue, réalisée sans transvasage, permettra enfin d'obtenir un tri dans l'ordre croissant de l'échantillon.

Echantillon vu par ordre croissant

-23.11722184	-20.52021828	-19.46032111	-19.28262783	-19.19134677
-18.75734992	-17.30971475	-17.14467035	-16.34011599	-15.16054654
-14.84830470	-14.76655916	-14.68792145	-14.41818323	-13.16301198
-12.92818039	-12.59212736	-12.12998379	-11.98010909	-11.90548743
-11.69397684	-11.45147589	-10.52718951	-10.52355401	-9.82497272
-9.64521447	-9.44664980	-9.43006685	-8.86287054	-8.86212592
-8.75742422	-8.73197068	-8.63164032	-8.60167513	-8.50888284
-8.44052148	-8.37893022	-8.31167025	-8.27074182	-8.18530187
-8.17116288	-8.08861273	-7.93059616	-7.66731069	-7.61357796
-7.58657528	-7.25834184	-6.99608957	-6.90087122	-6.89676472
-6.83498916	-6.83335346	-6.61637182	-5.91503132	-5.80017050
-5.77413565	-5.71359602	-5.62884370	-4.78288433	-4.37713352
-4.29400897	-4.25911260	-3.98616773	-3.91193490	-3.89058023
-3.73216365	-3.61998630	-3.60593051	-3.16249661	-3.15916844
-2.84698943	-2.78933727	-2.69576566	-2.52155685	-2.38930333
-2.18711432	-2.10084739	-2.06067266	-1.62206149	-1.51651527
-1.51226666	-1.42041743	-1.34735748	-1.23465467	-1.06194261
-1.05431724	-1.05425539	-.91103659	-.75139762	-.71723736
-.45114404	-.45062413	-.34639462	-.12508947	-.10507990
-.05280165	.06048231	.13953984	.26333193	.47323090
1.61432553	1.96126931	2.32278228	2.52895943	2.74777451
2.98500725	3.08246870	3.11200339	3.16440085	3.21531168
3.26717165	3.64402286	4.01962606	4.04084789	4.22643310
4.27607093	4.55878209	4.68049236	4.72571873	4.79198637
4.96646836	5.09234721	5.18197246	5.24352159	5.48238302
5.51073477	5.57673431	5.58953697	5.71304394	5.71930116
5.82942841	6.11640310	6.39741576	6.47937176	6.71197312
6.75430833	6.87681404	6.97045299	7.04651224	7.11623217
7.27778032	7.35599621	7.45819907	7.52512352	7.71088415
7.85032549	7.85460543	8.20978382	8.43610508	8.44621143
8.54214938	8.62360851	8.63149484	8.70235076	8.71496201
8.85777763	8.88742944	8.96830231	9.11767113	9.25952979
9.61159092	9.63181164	9.74893859	9.95543845	10.00444540
10.07298108	10.46384450	11.04463138	11.07407900	11.21260762
11.24194564	11.40581906	11.75425048	11.76480826	12.02142507
12.07604714	12.50355096	12.76356379	13.05193918	13.89971064
14.57536686	14.70087015	14.87262723	14.96246216	15.05692567
15.47603865	15.65402380	16.06010567	16.64963003	16.65242866
16.69873762	17.44671782	18.02794447	18.22375162	18.54519386
18.58665052	20.14959376	21.45119500	22.17653924	22.68388520

<<<<<FIN>>>>>

De la représentation d'un arbre de priorité équilibré dans une rangée découle d'un algorithme de tri intéressant. Cet algorithme peut-être comparé avec celui qui a été incorporé dans le programme *tris* du chapitre 5. Dans les deux cas, il s'agit d'algorithmes efficaces pour trier les éléments d'une rangée. Ce nouvel algorithme, le "heap-sort" par opposition au "quick-sort" du chapitre 5, se déroule en deux phases.

Dans la première,la rangée concernée est constituée en un arbre de priorité implicite ordonné selon la relation inverse de celle du tri; ainsi, si la rangée doit être triée dans l'ordre croissant de ses éléments, il sera formé un arbre de priorité implicite dont les composantes seront ordonnées de manière décroissante. Pour celà, les éléments de la rangée sont pris un à un, dans l'ordre croissant de leurs indices, et insérés dans l'arbre implicite constitué par l'ensemble des éléments qui le précèdent dans la rangée.

Dans la deuxième phase, l'arbre de priorité implicite est détruit par l'extraction successive de ses éléments qui sont alors stockés à leur emplacement définitif. Ceci marche puisque ce dernier emplacement est juste celui qui est libéré de l'arbre lors de l'opération qui en extrait l'élément correspondant.
Le listage suivant présente une version modifiée du programme *tris* ; seule la procédure *trier* a été changée: il y a été incorporé le "heap-sort" en lieu et place du "quick-sort".

```
tris                                        Vax Newton Compiler 0.2c11
                                                    Page    1
                 Source listing
     1  /* /*OLDSOURCE=USER2:[RAPIN]TRIS_PRIOR.NEW*/ */
     1  PROGRAM tris DECLARE
     4    real ROW vecteur;
     8
     8    Boolean FUNCTOR relation(real VALUE gauche,droite)VALUE
    19      croissant=BODY relation DO TAKE gauche<droite DONE,
    30      decroissant=BODY relation DO TAKE gauche>droite DONE,
    41      absolu=BODY relation DO TAKE ABS gauche<ABS droite DONE,
    54      classes_entieres=
    56        BODY relation DO TAKE FLOOR gauche<FLOOR droite DONE;
    67
    67    PROCEDURE trier
    69      (vecteur VALUE tab; relation VALUE inf)
    78     (*Rearrange les composantes du vecteur donne  tab  de maniere
    78      telle que l'on ait:
    78
    78        LOW tab<=j/\j<k/\k<=HIGH tab IMPL ~inf[tab[K],tab[j]]
    78
    78      Conditions d'emploi:
    78
    78      inf  denote une relation d'ordre (eventuellement partielle)
    78           sur les valeur reelles; pour  x, y, z  reels, on doit
    78           avoir:
    78
    78      inf[x,y] NAND inf[y,x] ;
    78      inf[x,y] NOR inf[y,x] IMPL (inf[x,z]==inf[y,z]) ;
    78      inf[x,y]/\inf[y,z] IMPL inf[x,z]
    78    *)
    78    DECLARE(*trier*)
    79
    79    MODULE t
    81      INDEX decalage
    83      ATTRIBUTE ordre
    85    DECLARE(*t*)
    86      integer VALUE ecart=PRED LOW tab;
    94
    94      integer VALUE ordre=HIGH tab-ecart;
   103      (*Le nombre d'elements de la rangee  tab *)
   103
   103      real ACCESS decalage
   106        (integer VALUE k)
   111      (*Le tableau  tab  vu avec un indicage de borne inferieure
   111        egale a  1 .
   111      *)
   111      DO TAKE tab[k+ecart] DONE
   120    DO(*t*)DONE;
   123
   123      integer VARIABLE dim:=1,pere,fils,aine,puine;
   137      real VARIABLE val
   140    DO(*trier*)
   141      (*Transforme la rangee en un arbre de priorite implicite;
   141        le critere de priorite est l'inverse de la relation
```

Source listing

```
 141       inf :  une valeur  x  sera prioritaire par rapport a  y
 141       si  inf[y,x]  est vrai
 141     *)
 141    UNTIL dim=ordre REPEAT
 146       (*On a un arbre implicite de dimension  dim ; y insere
 146         l'element  t[SUCC dim]
 146       *)
 146       val:=t[(dim:=SUCC dim)]; fils:=dim;
 162       WHILE TAKE
 164         IF fils>1 THEN
 169           inf[t[(pere:=fils%2)],val]
 184         DEFAULT FALSE DONE
 187       REPEAT t[fils]:=t[(fils:=pere)] REPETITION;
 203       t[fils]:=val
 209    REPETITION;
 211     (*Elimine un a un les elements de l'arbre implicite; les
 211       stocke a leur emplacement definitif
 211     *)
 211    UNTIL dim=1 REPEAT
 216       (*L'emplacement definitif de  t[1]  est en  t[dim] ;
 216         elimine cet element de l'arbre implicite et reinsere
 216         l'ancienne valeur de  t[dim]
 216       *)
 216       t[(pere:=1)]=:t[(PRED dim=:dim)]=:val;
 237       CYCLE descente REPEAT
 240
 240       IF
 241         (aine:=2*pere)>dim
 250       EXIT descente DONE;
 254
 254         fils:=
 256           IF aine=dim THEN aine ELSE
 263           IF inf[t[(puine:=SUCC aine)],t[aine]] THEN aine
 283           DEFAULT puine DONE;
 287
 287       UNLESS
 288         inf[val,t[fils]]
 297       EXIT descente DONE;
 301
 301         t[pere]:=t[(pere:=fils)]
 314       REPETITION;
 316       t[pere]:=val
 322    REPETITION
 323  DONE(*trier*);
 325
 325  PROCEDURE imprimer(vecteur VALUE vec) DO
 333    THROUGH vec INDEX k VALUE vk REPEAT
 340      IF k\5=1 THEN line DONE; edit(vk,12,8)
 358    REPETITION
 359  DONE(*imprimer*);
 361
 361  PROCEDURE traiter
```

```
tris                                              Vax Newton Compiler 0.2c11
                                                   Page    3
                Source listing

 363        (integer VALUE taille; real EXPRESSION terme;
 372         relation VALUE ordre)
 376      DECLARE(*traiter*)
 377        vecteur VALUE vec=
 381          THROUGH vecteur(1 TO taille):=terme REPETITION;
 392      DO(*traiter*)
 393        print(page,"***Vecteur original***");
 400        imprimer(vec);
 405        trier(vec,ordre);
 412        print(line,"***Vecteur trie***");
 419        imprimer(vec); print(line,"<<<FIN>>>")
 430      DONE(*traiter*)
 431    DO(*tris*)
 432      randomize;
 434      traiter(100,random,croissant);
 443      traiter(100,random,decroissant);
 452      traiter(50,normal,absolu);
 461      traiter(50,10*poisson,classes_entieres)
 471    DONE(*tris*)

    ****  No messages were issued  ****
```

Dans la nouvelle version de la procédure *trier*,on remarque le module *t* dont le rôle est de ramener à la valeur *1* la borne inférieure des indices de la rangée à trier.Cette convention facilite la navigation dans l'arbre de priorité implicite concerné.

Du fait du recours à la procédure *randomize*, il a été obtenu des résultats différents de ceux du chapitre 5: les opérations de tri ont porté sur d'autres échantillons aléatoires. Il apparaît clairement, par contre, que le programme est bien équivalent à celui du chapitre 5.

```
***Vecteur original***
    .88759637   .70366748   .35405242   .23198259   .19043387
    .33641416   .13691735   .56053349   .67354042   .22442732
    .68990752   .89598691   .22840488   .99786204   .56298938
    .57536595   .16795266   .09285346   .67751060   .51107415
    .32710342   .73573790   .61918043   .38873822   .03546898
    .47506910   .53847287   .25195965   .42102394   .62806806
    .89196468   .69082392   .57039007   .00202398   .30008400
    .30969776   .48210262   .17951115   .45740369   .99388031
    .39648842   .43806460   .97415471   .27986089   .95899747
    .07227256   .91972647   .28799911   .83721497   .51359451
    .82651929   .42627048   .94016480   .11813826   .03113007
    .37828040   .06122768   .61769031   .73384239   .83468913
    .00802252   .15577872   .09437347   .54367235   .85499965
    .79627938   .91473834   .39238469   .37457295   .57998659
    .70706715   .86789204   .41005658   .25729790   .58010772
    .77457321   .79094743   .59055749   .45513067   .48188553
    .11868213   .24778662   .88838893   .49760288   .52800467
    .42913120   .91664000   .45522907   .66028307   .67265771
    .93007548   .88927966   .40087074   .48376071   .53006908
    .61864589   .80419088   .80714670   .20676466   .43084603
***Vecteur trie***
    .00202398   .00802252   .03113007   .03546898   .06122768
    .07227256   .09285346   .09437347   .11813826   .11868213
    .13691735   .15577872   .16795266   .17951115   .19043387
    .20676466   .22442732   .22840488   .23198259   .24778662
    .25195965   .25729790   .27986089   .28799911   .30008400
    .30969776   .32710342   .33641416   .35405242   .37457295
    .37828040   .38873822   .39238469   .39648842   .40087074
    .41005658   .42102394   .42627048   .42913120   .43084603
    .43806460   .45513067   .45522907   .45740369   .47506910
    .48188553   .48210262   .48376071   .49760288   .51107415
    .51359451   .52800467   .53006908   .53847287   .54367235
    .56053349   .56298938   .57039007   .57536595   .57998659
    .58010772   .59055749   .61769031   .61864589   .61918043
    .62806806   .66028307   .67265771   .67354042   .67751060
    .68990752   .69082392   .70366748   .70706715   .73384239
    .73573790   .77457321   .79094743   .79627938   .80419088
    .80714670   .82651929   .83468913   .83721497   .85499965
    .86789204   .88759637   .88838893   .88927966   .89196468
    .89598691   .91473834   .91664000   .91972647   .93007548
    .94016480   .95899747   .97415471   .99388031   .99786204
<<<FIN>>>
```

A priori, si l'on doit trier les éléments d'une rangée selon une relation d'ordre donnée, on peut-être amené à choisir entre le "quick-sort" et le "heap-sort". La plupart des autres algorithmes que l'on rencontre parfois (tri par bulles, tri par insertion, algorithme de tri de Shell,...) sont moins efficaces ou alors, ceux qui sont efficaces (tri par fusion de listes, tri par baquets,...) consomment plus de mémoire. Il est donc intéressant d'indiquer les raisons qui peuvent conduire à préférer l'une ou l'autre de ces méthodes.

```
***Vecteur original***
    .24384692    .63763598    .84839529    .76417851    .62440537
    .56846965    .12301120    .91937994    .23278605    .02884898
    .22108928    .60031752    .19992798    .19722688    .85137104
    .90029309    .48939555    .35986283    .55243563    .36328222
    .49822843    .19670719    .50583193    .08836113    .25824712
    .20016997    .31767399    .93263289    .58875424    .63249975
    .30136721    .84399512    .40672099    .44843755    .12233535
    .72980222    .97912263    .23272049    .72936523    .77252374
    .19084492    .52085006    .87701586    .49738258    .97471240
    .91539844    .22922246    .03826058    .10051391    .39595406
    .26445758    .79816201    .71260106    .27368233    .32439725
    .20594331    .00346178    .37053782    .94002849    .18658503
    .95779780    .18127467    .36673995    .71191360    .99344408
    .40482563    .69246356    .42295752    .12261707    .16002086
    .53193256    .89515002    .65188659    .94802356    .36098023
    .06013053    .10093960    .20041727    .31931793    .32104141
    .61785065    .37656521    .34503807    .53847987    .32460309
    .08524144    .65816052    .41288099    .61281910    .39325773
    .90751675    .95417284    .73020882    .02243443    .21738757
    .33086458    .58843578    .48561251    .05537298    .92810847
***Vecteur trie***
    .99344408    .97912263    .97471240    .95779780    .95417284
    .94802356    .94002849    .93263289    .92810847    .91937994
    .91539844    .90751675    .90029309    .89515002    .87701586
    .85137104    .84839529    .84399512    .79816201    .77252374
    .76417851    .73020882    .72980222    .72936523    .71260106
    .71191360    .69246356    .65816052    .65188659    .63763598
    .63249975    .62440537    .61785065    .61281910    .60031752
    .58875424    .58843578    .56846965    .55243563    .53847987
    .53193256    .52085006    .50583193    .49822843    .49738258
    .48939555    .48561251    .44843755    .42295752    .41288099
    .40672099    .40482563    .39595406    .39325773    .37656521
    .37053782    .36673995    .36328222    .36098023    .35986283
    .34503807    .33086458    .32460309    .32439725    .32104141
    .31931793    .31767399    .30136721    .27368233    .26445758
    .25824712    .24384692    .23278605    .23272049    .22922246
    .22108928    .21738757    .20594331    .20041727    .20016997
    .19992798    .19722688    .19670719    .19084492    .18658503
    .18127467    .16002086    .12301120    .12261707    .12233535
    .10093960    .10051391    .08836113    .08524144    .06013053
    .05537298    .03826058    .02884898    .02243443    .00346178
<<<FIN>>>
```

```
***Vecteur original***
    .71726490   -.13298833 -1.99890051    .63151251   -.78511701
   -.62509815 -1.16982679    .34458951    .31709189 -1.91021871
    .10845035    .07316271 -2.08249786   -.79840127   -.51585229
    .66642431   -.37464908   -.42019136  2.07474840 -1.90087437
    .84977562 -1.42188800    .91811752   -.91482158  2.16553639
    .17587647    .88262733 -1.18282480 -1.20174847  1.22284004
  -1.44133938 -1.40651482    .07736374    .19944767    .42806684
    .25531034   -.14414152    .66103908   -.26327047    .92526881
    .86039488   -.71244434  2.01307036 -1.30748645    .94643778
   1.17458576 -1.21721353 -1.57937968 -1.69699219   -.21249446
***Vecteur trie***
    .07316271    .07736374    .10845035   -.13298833   -.14414152
    .17587647    .19944767   -.21249446    .25531034   -.26327047
    .31709189    .34458951   -.37464908   -.42019136    .42806684
   -.51585229   -.62509815    .63151251    .66103908    .66642431
   -.71244434    .71726490   -.78511701   -.79840127    .84977562
    .86039488    .88262733   -.91482158    .91811752    .92526881
    .94643778 -1.16982679  1.17458576 -1.18282480 -1.20174847
  -1.21721353  1.22284004 -1.30748645 -1.40651482 -1.42188800
  -1.44133938 -1.57937968 -1.69699219 -1.90087437 -1.91021871
  -1.99890051  2.01307036  2.07474840 -2.08249786  2.16553639
<<<FIN>>>
```

```
***Vecteur original***
  10.23381060    .25475474 11.42526236 11.21959444  4.13332591
   9.56770657  7.21207766  8.06913313 10.87631695  9.71546126
  26.58757057  2.99459725 20.82258861  3.17446398  9.02381280
   9.46336870    .81802536 14.50010568  9.22651803  1.93345263
   3.92246903 30.93381898    .15022367 23.20027700 30.29664438
    .74709375  5.56896378  4.28420411  8.46798119 20.41741306
  20.13408709  7.55973301 25.35189550    .26504259 17.59863652
  15.64857392 36.66436453  3.24162082 37.34549208  4.16874272
    .45755512    .66245821  9.76196295  4.08649235  7.09314822
   1.10848249    .89956188    .77727699  2.48982529  2.11313645
***Vecteur trie***
    .26504259    .81802536    .45755512    .74709375    .89956188
    .77727699    .66245821    .25475474    .15022367  1.93345263
   1.10848249  2.99459725  2.11313645  2.48982529  3.24162082
   3.92246903  3.17446398  4.08649235  4.16874272  4.13332591
   4.28420411  5.56896378  7.55973301  7.21207766  7.09314822
   8.06913313  8.46798119  9.46336870  9.22651803  9.76196295
   9.71546126  9.56770657  9.02381280 10.87631695 10.23381060
  11.42526236 11.21959444 14.50010568 15.64857392 17.59863652
  20.82258861 20.13408709 20.41741306 23.20027700 25.35189550
  26.58757057 30.93381898 30.29664438 36.66436453 37.34549208
<<<FIN>>>
```

On remarque d'abord que toutes deux permettent de trier une rangée de *n* éléments en un temps proportionnel à *n*ln(n)*.

<u>Quick-sort:</u>

Avantages:

- En général plus rapide que le "heap-sort" (le facteur de proportionnalité de *n*ln(n)* est plus petit).
- Sur une installation à processeurs multiples, peut se prêter à un calcul en parallèle (après la première application de la procédure *tri_sec*, les tris partiels sont indépendants et peuvent se dérouler simultanément).

Inconvénients:
- Présente occasionnellement des cas pathologiques ou le tri peut exiger un nombre d'opérations de l'ordre de *n**2*.
- Du fait de sa nature récursive, consomme en général plus de mémoire. Pour une rangée de *n* éléments, il est possible de s'assurer, moyennant quelques précautions, que la mémoire supplémentaire reste de l'ordre de *ln(n)*.

<u>Heap-sort:</u>

Avantages:
- Il n'y a pas de cas pathologiques.Quelque soit l'ordre initial des éléments de la rangée concernée, le tri sera réalisé en un temps borné par un multiple de *n*ln(n)*.
- L'algorithme est purement itératif. Il ne nécessite, en plus de la rangée à trier, qu'un nombre fixe d'emplacements de mémoire.

Inconvénients:
- En général plus lent que le "quick-sort".
- Ne se prête pas, de manière simple, à un calcul en parallèle sur une installation à processeurs multiples.

Chapitre 13

Simulation discrète

On est souvent amené à simuler, sur l'ordinateur, un système réel. Au chapitre 7, on a vu l'exemple de la simulation d'un jeu de dés. Très souvent, le facteur temps intervient explicitement : il s'agit de simuler l'évolution d'un système pendant une période donnée. Selon la nature de ce système, il y a deux façons principales de programmer une telle application, la simulation continue et la simulation discrète.

Dans le cas de la simulation continue, le système considéré évolue de manière continue au cours du temps; il est représenté au moyen d'une équation différentielle, d'une équation aux dérivées partielles ou d'un système de telles équations. L'ordinateur intervient alors pour résoudre numériquement ces équations; les algorithmes nécessaires pour le faire sont du ressort de l'analyse numérique : le sujet ne sera pas poursuivi dans ce cours.

Il est souvent commode de considérer que le système que l'on veut simuler ne change d'état, de manière macroscopique, qu'à des instants bien déterminés. Plus spécifiquement, on considère que les changements interviennent en une suite d'instants *t_1, t_2, t_3,...t_n...* .On parle alors de simulation discrète. Seule cette approche sera traitée ici.

Il n'est peut-être pas toujours évident si un système donné doit être simulé au moyen d'un modèle continu ou discret. Un certain bon sens peut cependant souvent suggérer la solution adéquate. Ainsi, la simulation de l'évolution de l'atmosphère terrestre en vue de l'établissement de prévisions météorologiques fera typiquement appel à un modèle continu. Connaissant les propriétés des gaz composant l'atmosphère, son évolution sera représentée au moyen d'équations aux dérivées partielles découlant des lois de la physique. Par contre, la simulation d'un système de guichets sera typiquement traitée au moyen d'un modèle discret. On admettra que le système change d'état chaque fois qu'un client nouveau se présente et chaque fois qu'il le quitte après avoir été servi. Des hypothèses de nature stochastique doivent être faites sur la fréquence d'arrivée de nouveaux clients (celle-ci peut d'ailleurs dépendre de l'heure) ainsi que sur le temps nécessaire pour servir un client. Les événements correspondants pourront dépendre de tirages aléatoires.

Dans un modèle de simulation discrète, on dénomme événement chaque changement d'état du modèle. Il convient d'établir une structure de données et de contrôle qui permette de gérer correctement l'ordonnancement des événements; cette structure est un échéancier. Un échéancier peut être construit sous la forme d'une queue de priorité de notices d'événement. Chaque notice d'événement consiste en une action future associée à l'heure où cette dernière doit intervenir; dans cette queue, les événements seront ordonnés selon les heures croissantes du déclenchement de l'action correspondante. Les actions pourront être programmées sous la forme de coroutines; ces dernières seront détachées sur un point d'arrêt adéquat tant qu'elles sont incorporées dans une notice d'événement. Les notices d'événement sont extraites de l'échéancier en fonction de l'heure de déclenchement de l'action correspondante; ce déclenchement sera provoqué en réactivant la coroutine (Fig. 46).

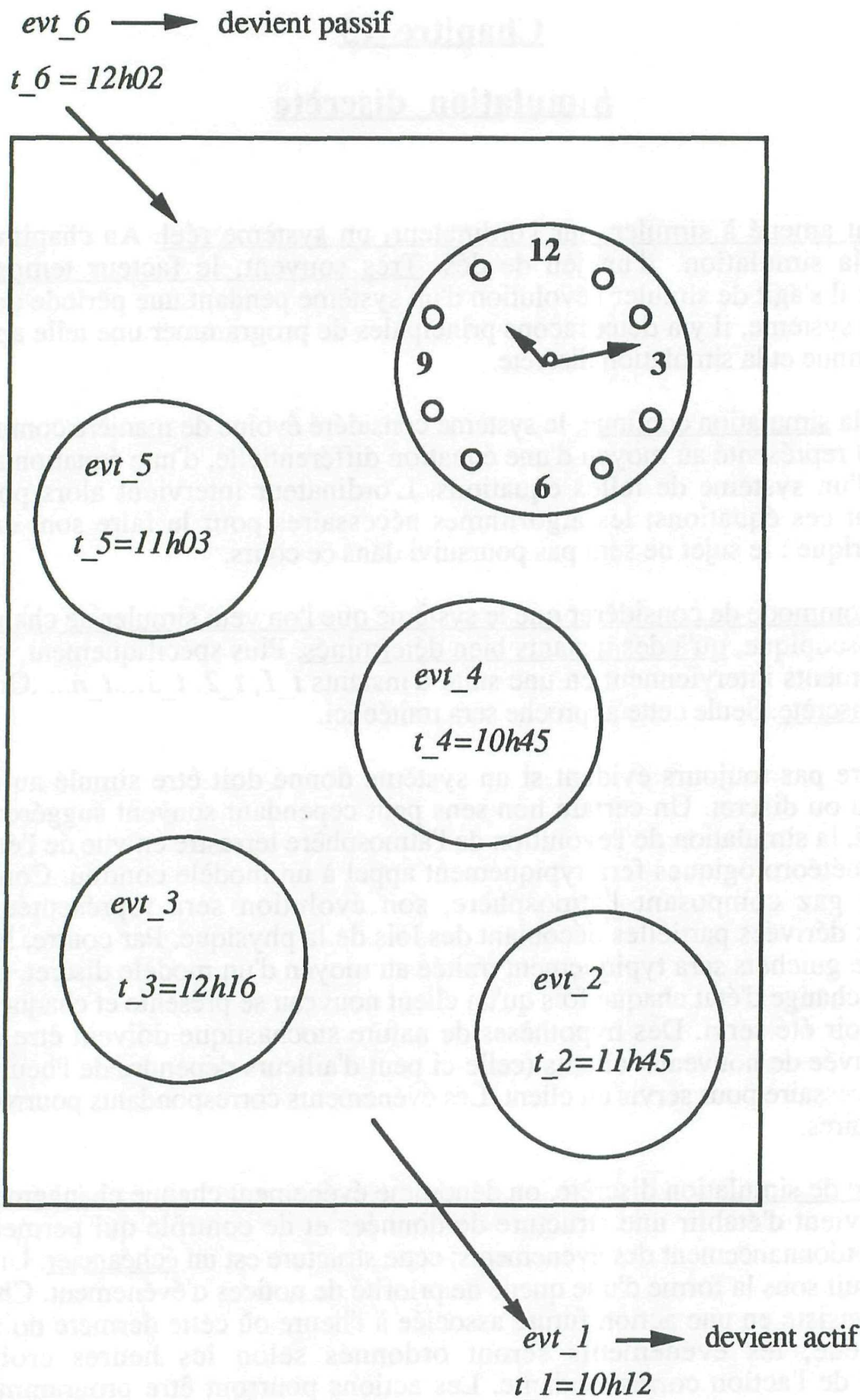


Fig. 46

A titre d'exemple, on a simulé un bureau contenant un ensemble *nombre_guichets = 10* guichets; la simulation portera sur une matinée entre les heures *heure_ouverture = 7h30* et *heure_fermeture = 11h45*. Pendant ces heures d'ouverture du bureau, des clients arrivent; chaque client effectuera un certain nombre de transactions : en moyenne, ce nombre est égal à *transactions_par_client = 5*. L'intervalle de temps entre l'arrivée de deux clients successifs est égal à *intervalle_moyen_arrivée_clients = 10 secs*. Parmi les guichets, il en est réservé *guichets_express = 2* pour les clients n'ayant pas plus de *transactions_expresses = 3* à accomplir, ceci dans le but d'assurer un service rapide à ces clients-là. Chaque guichet est doté d'un employé; certains employés sont plus efficaces que d'autres. Un employé "moyen" prend *temps_moyen_transaction = 20 secs* pour accomplir une transaction; selon les employés, ce

temps moyen peut varier entre *temps_moyen_minimum = 5 secs* et *temps_moyen_maximum = 30 secs* avec un écart-type de l'ordre de *écart_type_transaction = 5 secs.*

La simulation devra mettre en évidence l'évolution de la longueur des queues aux différents guichets au cours de la matinée. A la fin de la simulation, il sera indiqué le nombre de clients servis, les longueurs maximum et moyenne de la queue ainsi que les temps maximum et moyen entre le moment où un client est arrivé au bureau et celui où il l'a quitté après avoir été servi. La structure générale du programme est donnée à la figure 47.

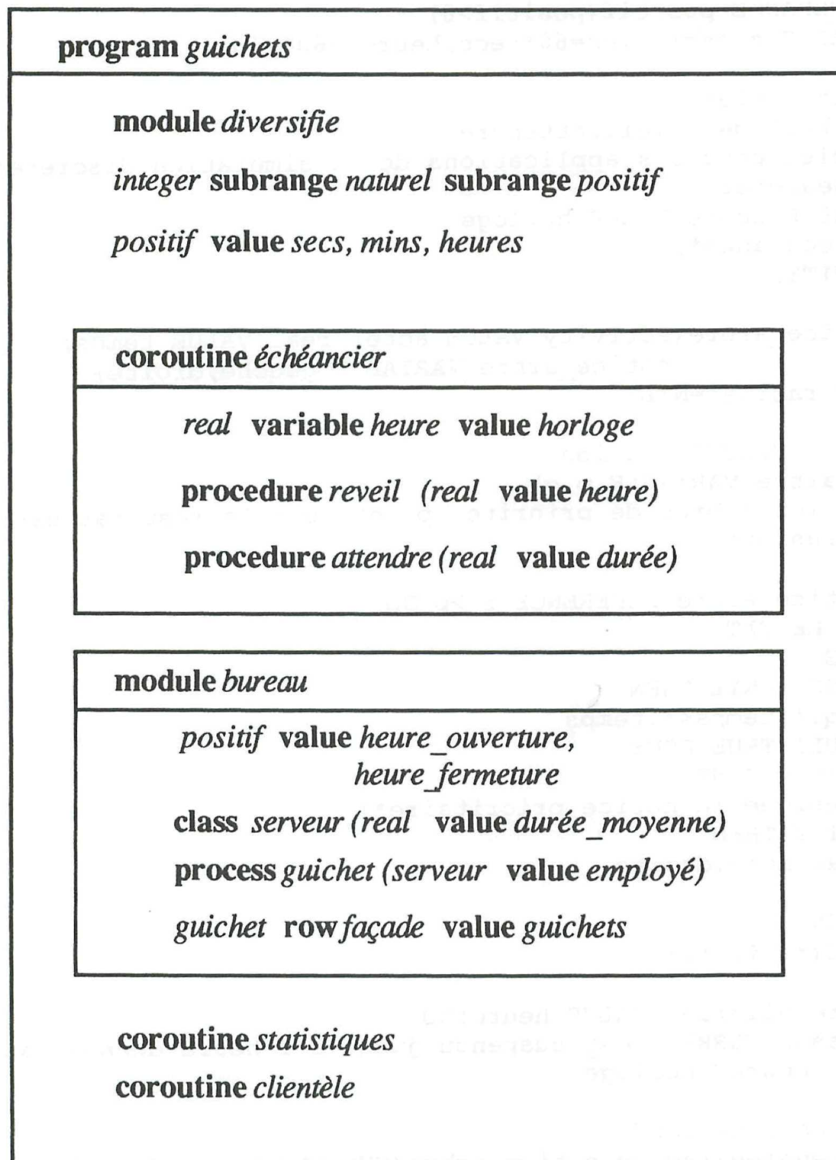


Fig. 47

Dans les définitions préliminaires, le module *diversifie* a pour seul rôle de réinitialiser le générateur aléatoire afin d'obtenir des simulations différentes à chaque exécution du programme. Dans ce genre d'application, il convient de choisir l'unité de temps avec laquelle seront faits les calculs; dans le cas présent, ce choix a porté sur la seconde. Les valeurs *secs*, *mins* et *heures* ont été introduites de manière à permettre une expression naturelle des temps exprimés dans une autre unité.

```
guichets                                        Vax Newton Compiler 0.2c13
                                    Page    1
                 Source listing

     1 /* /*OLDSOURCE=USER2:[RAPIN]GUICHETS.NEW*/ */
     1 PROGRAM guichets DECLARE
     4
     4 MODULE diversifie DO randomize DONE;
    10
    10    integer SUBRANGE naturel(naturel>=0)
    18            SUBRANGE positif(positif>0)
    25            VALUE secs=1,mins=60*secs,heures=60*mins;
    42
    42 COROUTINE echeancier
    44    ATTRIBUTE horloge,reveil,attendre
    50 (*Un echeancier pour les applications de la simulation discrete*)
    50 DECLARE(*echeancier*)
    51    real VARIABLE heure VALUE horloge
    56    (*L'heure courante*)
    56      :=-INFINITY;
    60
    60    OBJECT notice_arbre(activity VALUE acte; real VALUE temps;
    71                        notice_arbre VARIABLE gauche,droite)
    77      VARIABLE racine:=NIL;
    82
    82    notice_arbre FUNCTION union
    85       (notice_arbre VARIABLE p,q)
    92    (*Fusionne les arbres de priorite  p  et  q ; le resultat est
    92      l'arbre fusionne.
    92    *)
    92    DECLARE notice_arbre REFERENCE r->p DO
    99      WITHIN q REPEAT
   102        IF TAKE
   104          UNLESS r=NIL THEN
   109            (*q.*)temps<r.temps
   114          DEFAULT TRUE DONE
   117        THEN r:=:q DONE;
   123        (* r designe la notice prioritaire*)
   123        CONNECT r THEN
   126          r->gauche:=:droite
   131        DONE
   132      REPETITION
   133    TAKE p DONE(*union*);
   137
   137    PROCEDURE reveil(real VALUE heure)DO
   145    (*Le processus CURRENT est suspendu jusqu'a l'heure donnee; sans
   145      effet si  heure<=horloge
   145    *)
   145      IF heure>horloge THEN
   150        racine:=union(racine,notice_arbre(CURRENT,heure,NIL,NIL))
   167      RETURN DONE
   169    DONE(*reveil*);
   171
   171    PROCEDURE attendre(real VALUE duree)DO
   179   (*Le preocessus CURRENT est suspendu pendant la duree donnee; sans
   179      effet si  duree<0
```

Le rôle de la coroutine *échéancier* a déjà été expliqué. On remarque que les notices d'événement y sont structurées en un arbre de priorité issu de la variable *racine*. On y a incorporé les deux primitives *réveil* et *attendre*. Selon l'application, d'autres primitives analogues pourraient se révéler utiles; on pourrait, par exemple, penser à une procédure *planifier* de la forme :

procedure *planifier*
    (activity **value** *acte; real* **value** *heure)*
*(** Fait intervenir, au temps *heure*, la
    prochaine phase d'activité du processus
    acte donné.

    Condition d'emploi :
        state *acte = detached* ∧ *horloge* < = *heure*
**)*

La partie exécutable de l'échéancier est le moteur de la simulation. Initialement détachée, elle est réactivée après l'insertion des notices d'événement prévisibles dès le début de la simulation. C'est sous son contrôle que la simulation sera ensuite accomplie. Lorsque l'échéancier ne contient plus aucune notice d'événement, la simulation est achevée : l'échéancier achève alors son exécution.

La classe *serveur* et le processus *guichet* dénotent les composantes principales du module *bureau*. On remarque qu'il s'est révélé plus commode de rendre actifs les guichets et passifs les serveurs, contrairement à ce qu'aurait suggéré l'intuition. Serveurs et guichets contiennent les variables au moyen desquelles seront récoltées les informations statistiques souhaitées. Un serveur a la structure de la figure 48.

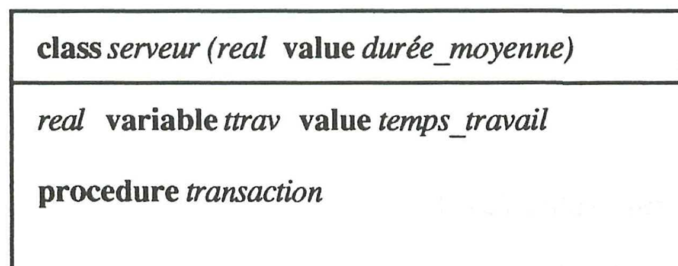


Fig. 48

Un client active la procédure *transaction* pour chacune des transactions qu'il doit accomplir. Les temps nécessaires aux transactions individuelles sont tirées selon une distribution exponentielle négative, selon l'hypothèse statistique (relativement plausible) que la majorité des transactions seront de courte durée tandis qu'un petit nombre de transactions pourront être très longues. La structure d'un guichet est donnée à la figure 49.

process *guichet (serveur* **value** *employé)*

---

positif **variable** *trsmax* **value** *nombre_max_transactions*

guichet **function** *plafonne_transactions*
(positif **value** *transactions)*

activity **queue** *file_attente* **variable** *clients*

naturel **expression** *longueur_queue*

naturel **variable** *lmax* **value** *longueur_maximum*

real **variable** *dernier_mouvement*

real **variable** *tpsmax* **value** *attente_maximum*

real **variable** *tpsom*

naturel **variable** *nclts* **value** *nombre_clients*

real **expression** *attente_moyenne*

real **variable** *lgtps*

procedure *entrer*

procedure *quitter (real* **value** *durée)*

real **variable** *ferme* **value** *fermeture*

real **variable** *lgmoy* **value** *longueur_moyenne*

Fig 49

```
guichets                                          Vax Newton Compiler 0.2c13
                                           Page    2
                     Source listing

      *)
        UNLESS duree<0 THEN
      racine:=union(racine,notice_arbre(CURRENT,horloge+duree,NIL,NIL))
        RETURN DONE
      DONE(*attendre*)
    DO(*echeancier*)RETURN
      WITHIN racine REPEAT
        (*Enleve de l'arbre de priorite la prochaine notice d'evenement
          a prendre en compte, avance l'horloge et effectue l'action
          correspondante.
        *)
        heure:=temps; racine:=union(gauche,droite);
        ACTIVATE acte NOW
      REPETITION;
      (*La simulation est terminee*)
      heure:=INFINITY
    DONE(*echeancier*);
    /* /*EJECT*/ */
```

Lorsqu'il arrive à un guichet, un client y effectue la procédure *entrer*; lorsqu'il le quitte, il fait exécuter la primitive *quitter* en lui communiquant le temps écoulé depuis le moment où il est arrivé. On remarque l'usage de la variable *lgtps* pour calculer, à la fin de la simulation, la longueur moyenne de la file d'attente associée au guichet : cette variable a pour valeur la somme cumulée de la longueur de la file multipliée par la durée pendant laquelle elle a eu la longueur considérée. Le lecteur examinera attentivement la structure de contrôle incorporée dans la partie exécutable des processus *guichet*, en conjonction avec les primitives de contrôle incorporées dans les procédures *entrer* et *quitter*. Au moyen de clauses **resume**, client et guichet se transmettent le contrôle lorsqu'un client arrive à un guichet vide et chaque fois qu'un client quitte son guichet. A la fin de la simulation, les variables *ferme* et *lgmoy* sont mises à jour. On remarque que ceci impliquera une réactivation du guichet à l'*heure_fermeture* si la queue du guichet est vide à ce moment-là; dans le cas contraire, le dernier client provoquera automatiquement cette mise à jour au moment de quitter le guichet.

```
guichets                                    Vax Newton Compiler 0.2c13
                                         Page    3
                 Source listing

MODULE bureau
 ATTRIBUTE heure_ouverture,heure_fermeture,serveur,guichet,guichets
DECLARE(*bureau*)
  positif VALUE heure_ouverture=7*heures+30*mins,
                heure_fermeture=11*heures+45*mins;

  CLASS serveur
    ATTRIBUTE duree_moyenne,transaction,temps_travail
    (real VALUE duree_moyenne)
  (*Modelise un employe; il lui faut, en moyenne, le temps
    duree_moyenne  pour accomplir une transaction.
  *)
  DECLARE(*serveur*)
    real VARIABLE ttrav VALUE temps_travail
    (*Le temps total pendant lequel le serveur a travaille*)
      :=0;

    PROCEDURE transaction
    (*Simule une transaction du client CURRENT*)
    DECLARE real VALUE temps_transaction=duree_moyenne*poisson DO
      attendre(temps_transaction);
      ttrav:=ttrav+temps_transaction
    DONE(*transaction*)
  DO(*serveur*)DONE;

  PROCESS guichet
    VALUE ce_guichet
    ATTRIBUTE
      employe,nombre_max_transactions,plafonne_transactions,
      longueur_queue,longueur_maximum,longueur_moyenne,
      nombre_clients,entrer,quitter,
      fermeture,attente_maximum,attente_moyenne
    (serveur VALUE employe)
  (*Un guichet desservi par un serveur donne  employe .A ce guichet
    est associe une file d'attente de clients. A son tour, chaque
    client effectuera un certain nombre de transactions; ce nombre
    est limite par la valeur  nombre_max_transactions .
  *)
  DECLARE(*guichet*)
positif VARIABLE trsmax VALUE nombre_max_transactions:=integer MAX;

    guichet FUNCTION plafonne_transactions
      (positif VALUE transactions)
   (*Limite a  transactions  le nombre maximum de transactions qu'un
      client peut faire a ce guichet. Le resultat est le guichet
      concerne.
    *)
    DO trsmax:=transactions TAKE ce_guichet DONE;

    CONSTANT capac_init=20,incr_fac=1.5;

    activity QUEUE file_attente
```

```
guichets                                        Vax Newton Compiler 0.2c13
                                          Page    4
                Source listing

            VARIABLE clients:=file_attente(capac_init);

          naturel EXPRESSION longueur_queue=
          (*Le nombre de clients en attente a ce guichet*)
            CARD clients;

          naturel VARIABLE lmax VALUE longueur_maximum
          (*La longueur maximum de la file d'attente a ce guichet*)
            :=0;

          real VARIABLE dernier_mouvement:=horloge;
          (*L'heure a laquelle est intervenu la derniere entree ou sortie
            de client.
          *)
          real VARIABLE tpsmax VALUE attente_maximum
          (*Le temps maximum qu'un client a passe au guichet*)
            :=0;

          real VARIABLE tpsom
          (*La somme des temps d'attente des clients*)
            :=0;

          naturel VARIABLE nclts VALUE nombre_clients
          (*Le nombre de clients ayant utilise ce guichet*)
            :=0;

          real EXPRESSION attente_moyenne=
          (*Le temps moyen qu'un client a passe au guichet.

            Condition d'emploi:  nombre_clients>0
          *)
            tpsom/nombre_clients;

          real VARIABLE lgtps
          (*La somme des longueurs de la queue multipliee par le temps
            que celle-ci a eu la longueur correspondante.
          *)
            :=0;

          PROCEDURE entrer DO
          (*Modelise l'arrivee du client CURRENT a ce guichet*)
        lgtps:=lgtps+CARD clients*(horloge-(horloge=:dernier_mouvement));
            IF FULL clients THEN
              THROUGH
                (file_attente(CAPACITY clients*incr_fac)=:clients)
                VALUE client
              REPEAT clients APPEND client REPETITION
            DONE;
            clients APPEND CURRENT;
            lmax:=lmax MAX CARD clients;
            nclts:=SUCC nclts;
            IF CARD clients=1 THEN
```

```
        ce_guichet RESUME
      DEFAULT RETURN DONE
    DONE(*entrer*);

    PROCEDURE quitter
      (real VALUE duree)
    (*Modelise le depart du client CUNRENT ; celui-ci a passe
      le temps  duree  donne au guichet.
    *)
    DECLARE activity VARIABLE client DO
      tpsmax:=tpsmax MAX duree; tpsom:=tpsom+duree;
  lgtps:=lgtps+CARD clients*(horloge-(horloge=:dernier_mouvement));
      client FROM clients;
      ce_guichet RESUME
    DONE(*quitter*);

    real VARIABLE ferme VALUE fermeture;
    (*L'heure a laquelle le dernier client a quitte le guichet.

      Condition d'emploi:  la simulation doit etre terminee
    *)

    real VARIABLE lgmoy VALUE longueur_moyenne
    (*La longueur moyenne de la file d'attente.

      Condition d'emploi:  la simulation doit etre terminee
    *)
  DO(*guichet*)
    WHILE
      UNTIL EMPTY clients REPEAT
        clients FRONT RESUME
      REPETITION
    TAKE
      horloge<heure_fermeture
    REPEAT RETURN REPETITION;
    lgmoy:=lgtps/((ferme:=horloge)-heure_ouverture)
  DONE(*guichet*);

  CONSTANT nombre_guichets=10,
           guichets_express=2,transactions_expresses=3;
  real VALUE temps_moyen_transaction=20*secs,
             ecart_type_transaction=5*secs,
             temps_moyen_minimum=5*secs,
             temps_moyen_maximum=30*secs;

  guichet ROW facade VALUE guichets=
    THROUGH
      facade(1 TO nombre_guichets)
      INDEX k  REFERENCE guichet_k
      :=guichet
          (serveur
           (temps_moyen_minimum MAX
```

La coroutine *statistiques* est responsable de l'ensemble de la mise en place des résultats. Après l'impression du titre initial, elle s'insère dans l'échéancier pour se réveiller chaque fois que l'on désire examiner l'état des files d'attente. A l'*heure_fermeture*, cette coroutine réactive les guichets dont la file d'attente est vide, afin de leur faire achever leur partie exécutable. Il est ensuite élaboré la clause **exchange**. Du point de vue syntaxique, cette primitive de manipulation de coroutines sépare, au même titre que le point-virgule ou le **return**, deux instructions consécutives d'une suite d'énoncés. L'effet est de réattacher la coroutine qui exécute cette clause à la suite de celle à laquelle elle était attachée. Ceci implique que l'exécution des énoncés qui suivent le symbole **exchange** n'interviendra qu'après l'élaboration de cette dernière coroutine. Dans le cas présent, on vérifie que la coroutine *statistiques* est attachée à l'échéancier au moment d'élaborer la clause **exchange**; l'échéancier va alors reprendre le contrôle : il fera achever l'exécution des clients encore dans les files d'attente à l'heure de fermeture du bureau. La simulation sera achevée lorsque tous les clients auront quitté le système. L'échéancier achève alors son exécution; ensuite la coroutine *statistiques* reprend la sienne pour imprimer les statistiques cumulées, pour chaque guichet, sur l'ensemble de la simulation.

```
guichets                                        Vax Newton Compiler 0.2c13
                                         Page     6
                     Source listing

                    temps_moyen_transaction+ecart_type_transaction*normal
                    MIN temps_moyen_maximum))
            REPEAT
              UNLESS k>guichets_express THEN
                guichet_k.plafonne_transactions(transactions_expresses)
              DONE
            REPETITION
          DO(*bureau*)DONE;
          /* /*EJECT*/ */
```

```
guichets                                    Vax Newton Compiler 0.2c13
                                 Page    7
                          Source listing

   COROUTINE statistiques DECLARE
     positif VALUE intervalle_tabulation=5*mins
   DO(*statistiques*)
     print("*****Simulation d'un ensemble de guichets*****",line,line,
           "***Evolution des files d'attente***",line,line);
     print("Heure"); column(10);
     THROUGH guichets INDEX num REPEAT edit(num,5,0) REPETITION;
     line; line;
     FOR integer VALUE h
       FROM heure_ouverture BY intervalle_tabulation TO heure_fermeture
     REPEAT
       reveil(h);
       edit(h%heures,2,0); print("H");
       edit(h\heures%mins,2,0); print(":"); column(10);
       THROUGH guichets VALUE guiche REPEAT
         edit(guiche.longueur_queue,5,0)
       REPETITION;
       line
     REPETITION;
     print("***FIN***",page);
     THROUGH guichets VALUE guiche REPEAT
       IF guiche.longueur_queue=0 THEN
         ACTIVATE guiche NOW
       DONE
     REPETITION
     (*Attend que tout soit termine*)
   EXCHANGE
     (*Imprime les statistiques completes*)
     THROUGH
       guichets  INDEX k  VALUE guichet_k
     REPEAT
       print("***Guichet numero:"_,edit(k,2,0),"***",line);
       CONNECT guichet_k THEN
         print(___,"Heure de fermeture du guichet:"_,
                   edit(fermeture%heures,2,0),"H",
                   edit(fermeture\heures/mins,5,2),"mins",line,
               ___,"Taux d'occupation de l'employe:"_,
                   edit(100*employe.temps_travail/
                        (fermeture-heure_ouverture),5,1),"%",line,
               ___,"Temps moyen pour traiter une transaction:"_,
                   edit(employe.duree_moyenne,4,1),"secs",line,
               ___,"Nombre de clients servis:"_,
                   edit(nombre_clients,5,0),line,
               ___,"Attente maximum:"_,
                   edit(attente_maximum/mins,6,2),"mins",line,
               ___,"Attente moyenne:"_,
                   edit(attente_moyenne/mins,6,2),"mins",line,
               ___,"Longueur maximum de la queue:"_,
                   edit(longueur_maximum,3,0),line,
               ___,"Longueur moyenne de la queue:"_,
                   edit(longueur_moyenne,6,2),line,
               "***FIN***",line,line)
```

La coroutine *clientèle* se place dans l'échéancier jusqu'à l'heure d'ouverture du bureau. Jusqu'à l'heure de fermeture, elle met ensuite en oeuvre l'arrivée des clients successifs. L'intervalle de temps entre l'arrivée de deux clients consécutifs est tiré selon une distribution exponentielle négative. Ceci se justifie, statistiquement, si l'on suppose que les clients sont indépendants les uns des autres; la distribution exponentielle négative jouit en effet de la propriété suivante : le temps qu'il faudra, en moyenne, attendre jusqu'au prochain événement ne dépend pas du temps qui s'est écoulé depuis le dernier événement.

L'algorithme des clients, incorporé dans le type processus *client* est simple. Après avoir noté son heure d'arrivée, un client choisit son guichet. Il prend celui dont la file d'attente est la plus courte, parmi ceux qu'il a le droit d'utiliser; si plusieurs guichets ont des files de même longueur, il choisit le premier. Pour le reste, le client s'appuye sur les primitives définies dans le processus *guichet* et la classe *serveur*.

La partie exécutable du programme ***guichets*** ne fait que mettre en oeuvre l'échéancier; initialement, ce dernier contient deux notices d'événement, l'une pour la coroutine *statistiques* et l'autre pour la coroutine *clientèle*. Tout le reste se déroule sous le contrôle de l'échéancier.

```
guichets                                    Vax Newton Compiler 0.2c13
                                          Page      8
                     Source listing

         DONE(*CONNECT guichet_k*)
       REPETITION;
       print("<<<<<FIN DE LA SIMULATION>>>>>")
     DONE(*statistiques*);
     /* /*EJECT*/ */
```

```
guichets                                          Vax Newton Compiler 0.2c13
                                      Page    9
                 Source listing


 1082  COROUTINE clientele DECLARE
 1085    CONSTANT transactions_par_client=5;
 1090    positif VALUE intervalle_moyen_arrivee_clients=10*secs;
 1098
 1098    PROCESS client DECLARE
 1101    (*Modelise l'algorithme de chaque client*)
 1101      real VALUE heure_arrivee=horloge;
 1107    positif VALUE nombre_trans=CEIL(transactions_par_client*poisson);
 1118      guichet VALUE mon_guichet=
 1122        DECLARE
 1123          guichet VARIABLE guiche:=NONE;
 1129          naturel VARIABLE long:=integer MAX
 1135        DO
 1136          THROUGH
 1137            guichets VALUE g
 1140          REPEAT
 1141            UNLESS nombre_trans>g.nombre_max_transactions THEN
 1148              IF g.longueur_queue<long THEN
 1155                guiche:=g; long:=g.longueur_queue
 1164              DONE
 1165            DONE
 1166          REPETITION
 1167        TAKE guiche DONE
 1170    DO(*client*)
 1171      mon_guichet.entrer;
 1175      FOR integer FROM 1 TO nombre_trans REPEAT
 1182        mon_guichet.employe.transaction
 1187      REPETITION;
 1189      mon_guichet.quitter(horloge-heure_arrivee)
 1197    DONE(*client*)
 1198  DO(*clientele*)
 1199    reveil(heure_ouverture);
 1204    WHILE
 1205      attendre(intervalle_moyen_arrivee_clients*poisson)
 1211    TAKE horloge<heure_fermeture REPEAT client REPETITION
 1218  DONE(*clientele*)
 1219  /* /*EJECT*/ */




guichets                                          Vax Newton Compiler 0.2c13
                                      Page   10
                 Source listing


 1219  DO(*guichets*)
 1220    ACTIVATE echeancier NOW
 1223  DONE(*guichets*)

       ****  No messages were issued  ****
```

*****Simulation d'un ensemble de guichets*****

Evolution des files d'attente

Heure	1	2	3	4	5	6	7	8	9	10
7H30:	0	0	0	0	0	0	0	0	0	0
7H35:	0	1	1	2	2	2	2	2	0	1
7H40:	1	0	3	2	2	2	2	2	2	2
7H45:	1	0	3	2	2	2	2	3	3	2
7H50:	2	0	3	4	3	3	2	3	3	3
7H55:	2	0	4	4	3	3	3	3	3	3
8H 0:	0	0	5	5	4	4	4	3	4	4
8H 5:	0	1	6	5	4	5	4	5	5	5
8H10:	2	0	5	5	5	4	5	4	4	4
8H15:	1	0	6	5	6	6	5	5	5	5
8H20:	1	1	6	7	6	6	5	5	6	5
8H25:	2	1	7	6	6	6	6	6	5	6
8H30:	0	1	7	7	7	7	7	6	6	5
8H35:	0	1	8	8	8	7	7	7	7	7
8H40:	1	0	10	10	10	9	9	10	9	9
8H45:	2	2	9	9	9	9	9	8	8	9
8H50:	2	2	10	10	10	10	10	10	10	9
8H55:	0	0	11	12	12	12	12	11	12	11
9H 0:	1	1	12	12	12	12	11	11	11	12
9H 5:	0	1	13	13	13	13	12	12	12	12
9H10:	1	0	14	14	14	13	14	13	12	13
9H15:	0	0	14	15	15	14	14	14	14	14
9H20:	3	2	15	15	14	14	15	14	15	15
9H25:	2	0	16	16	16	16	16	15	15	15
9H30:	2	1	16	17	17	17	17	17	16	16
9H35:	1	1	17	17	17	17	17	16	17	17
9H40:	0	0	18	18	18	18	18	18	18	18
9H45:	2	0	19	19	19	19	19	19	18	18
9H50:	1	1	19	19	19	19	18	18	18	18
9H55:	0	0	19	19	19	18	18	18	18	18
10H 0:	3	2	20	20	19	19	19	19	19	19
10H 5:	1	0	18	20	18	17	17	17	17	17
10H10:	2	0	18	18	18	17	17	17	17	18
10H15:	2	2	19	18	18	18	18	18	18	18
10H20:	1	0	19	18	18	18	18	18	18	18
10H25:	1	1	18	18	18	17	18	16	18	17
10H30:	2	0	18	17	18	18	18	18	18	18
10H35:	1	0	18	18	18	18	17	17	18	17
10H40:	1	1	18	18	18	17	18	18	17	17
10H45:	2	0	20	20	20	19	19	19	19	19
10H50:	2	1	20	19	20	20	19	18	18	19
10H55:	1	0	21	21	21	21	20	20	20	20
11H 0:	0	2	21	20	21	19	20	20	20	19
11H 5:	2	1	21	21	21	21	20	20	20	20
11H10:	2	2	22	22	22	22	22	21	21	21
11H15:	1	1	23	23	23	23	22	22	22	22
11H20:	0	0	23	23	22	23	23	23	23	22
11H25:	1	1	25	25	25	25	25	24	24	24
11H30:	3	2	26	24	25	25	25	24	25	25
11H35:	2	2	26	26	26	24	25	24	25	25
11H40:	0	1	27	26	26	26	26	26	26	26
11H45:	2	2	27	26	27	27	27	27	27	27

FIN

```
***Guichet numero:  1***
   Heure de fermeture du guichet: 11H46.28mins
   Taux d'occupation de l'employe:  68.4%
   Temps moyen pour traiter une transaction: 14.9secs
   Nombre de clients servis:   381
   Attente maximum:   3.60mins
   Attente moyenne:    .72mins
   Longueur maximum de la queue:  5
   Longueur moyenne de la queue:  1.07
***FIN***

***Guichet numero:  2***
   Heure de fermeture du guichet: 11H45.46mins
   Taux d'occupation de l'employe:  51.6%
   Temps moyen pour traiter une transaction: 14.5secs
   Nombre de clients servis:   309
   Attente maximum:   2.10mins
   Attente moyenne:    .61mins
   Longueur maximum de la queue:  4
   Longueur moyenne de la queue:   .74
***FIN***

***Guichet numero:  3***
   Heure de fermeture du guichet: 13H35.74mins
   Taux d'occupation de l'employe:  99.7%
   Temps moyen pour traiter une transaction: 25.4secs
   Nombre de clients servis:    96
   Attente maximum: 114.33mins
   Attente moyenne:  56.18mins
   Longueur maximum de la queue: 28
   Longueur moyenne de la queue: 14.74
***FIN***

***Guichet numero:  4***
   Heure de fermeture du guichet: 13H  .17mins
   Taux d'occupation de l'employe:  99.6%
   Temps moyen pour traiter une transaction: 22.9secs
   Nombre de clients servis:    94
   Attente maximum:  89.40mins
   Attente moyenne:  50.73mins
   Longueur maximum de la queue: 27
   Longueur moyenne de la queue: 14.44
***FIN***

***Guichet numero:  5***
   Heure de fermeture du guichet: 13H 6.89mins
   Taux d'occupation de l'employe:  99.4%
   Temps moyen pour traiter une transaction: 20.4secs
   Nombre de clients servis:   116
   Attente maximum:  85.47mins
   Attente moyenne:  40.77mins
   Longueur maximum de la queue: 27
   Longueur moyenne de la queue: 14.04
***FIN***

***Guichet numero:  6***
   Heure de fermeture du guichet: 12H29.37mins
   Taux d'occupation de l'employe:  99.1%
   Temps moyen pour traiter une transaction: 15.9secs
   Nombre de clients servis:   143
```

```
   Attente maximum:  48.78mins
   Attente moyenne:  29.74mins
   Longueur maximum de la queue:  27
   Longueur moyenne de la queue:  14.21
***FIN***

***Guichet numero:  7***
   Heure de fermeture du guichet: 13H17.09mins
   Taux d'occupation de l'employe:  99.1%
   Temps moyen pour traiter une transaction: 22.8secs
   Nombre de clients servis:    99
   Attente maximum:  94.46mins
   Attente moyenne:  49.77mins
   Longueur maximum de la queue:  27
   Longueur moyenne de la queue:  14.20
***FIN***

***Guichet numero:  8***
   Heure de fermeture du guichet: 12H55.44mins
   Taux d'occupation de l'employe:  99.0%
   Temps moyen pour traiter une transaction: 20.7secs
   Nombre de clients servis:   122
   Attente maximum:  72.69mins
   Attente moyenne:  37.47mins
   Longueur maximum de la queue:  27
   Longueur moyenne de la queue:  14.05
***FIN***

***Guichet numero:  9***
   Heure de fermeture du guichet: 13H33.57mins
   Taux d'occupation de l'employe:  98.9%
   Temps moyen pour traiter une transaction: 23.3secs
   Nombre de clients servis:   101
   Attente maximum: 109.19mins
   Attente moyenne:  48.49mins
   Longueur maximum de la queue:  27
   Longueur moyenne de la queue:  13.47
***FIN***

***Guichet numero: 10***
   Heure de fermeture du guichet: 13H16.16mins
   Taux d'occupation de l'employe:  98.9%
   Temps moyen pour traiter une transaction: 24.3secs
   Nombre de clients servis:    99
   Attente maximum:  91.66mins
   Attente moyenne:  48.81mins
   Longueur maximum de la queue:  27
   Longueur moyenne de la queue:  13.96
***FIN***

<<<<<FIN DE LA SIMULATION>>>>>
```

*****Simulation d'un ensemble de guichets*****

Evolution des files d'attente

Heure	1	2	3	4	5	6	7	8	9	10
7H30:	0	0	0	0	0	0	0	0	0	0
7H35:	1	1	1	1	1	1	1	0	0	0
7H40:	1	0	1	1	1	1	1	1	1	1
7H45:	0	1	1	1	1	1	1	0	1	1
7H50:	1	1	2	1	1	0	1	1	0	0
7H55:	0	0	1	1	1	1	1	1	1	0
8H 0:	1	0	2	2	1	1	1	1	1	1
8H 5:	2	2	2	2	1	1	1	2	1	0
8H10:	1	1	2	1	1	1	1	1	1	1
8H15:	2	1	2	2	1	1	1	1	1	1
8H20:	0	0	0	1	1	1	0	0	1	0
8H25:	1	0	2	2	1	2	1	1	1	1
8H30:	0	1	1	1	1	1	1	0	1	0
8H35:	1	1	2	2	2	2	1	2	1	0
8H40:	1	1	2	2	1	2	1	1	2	1
8H45:	1	2	2	2	2	1	2	1	2	2
8H50:	3	3	2	3	3	3	3	3	3	2
8H55:	1	2	4	4	3	3	3	3	3	3
9H 0:	1	2	4	3	3	3	2	2	3	2
9H 5:	1	1	2	1	1	1	1	0	1	0
9H10:	0	1	2	2	1	0	1	1	1	1
9H15:	2	1	2	2	2	2	2	1	1	1
9H20:	0	1	1	1	1	1	1	1	1	1
9H25:	2	2	2	2	2	1	1	0	1	1
9H30:	1	2	1	2	2	1	1	1	0	0
9H35:	1	1	1	2	2	2	1	1	1	1
9H40:	2	0	3	2	1	2	2	1	0	1
9H45:	1	1	2	0	0	0	1	0	1	0
9H50:	1	1	0	0	1	1	0	1	1	0
9H55:	1	1	1	1	0	0	1	1	0	0
10H 0:	1	0	1	1	1	1	0	0	1	0
10H 5:	1	0	2	2	2	2	2	2	1	1
10H10:	1	1	1	1	1	1	1	1	1	0
10H15:	1	1	0	1	0	1	1	1	0	0
10H20:	0	1	1	1	0	1	1	0	0	1
10H25:	0	0	1	2	1	1	1	1	1	1
10H30:	1	2	2	2	1	1	0	0	1	1
10H35:	1	1	2	2	2	2	1	1	1	1
10H40:	1	1	2	3	1	1	1	0	1	0
10H45:	0	1	2	2	1	1	0	1	1	1
10H50:	2	1	2	1	1	1	1	1	1	1
10H55:	1	0	2	2	2	1	2	2	1	1
11H 0:	0	1	2	2	0	1	2	1	1	1
11H 5:	2	2	2	1	1	1	2	1	1	1
11H10:	0	0	2	1	0	0	1	1	1	1
11H15:	0	1	1	1	1	2	1	1	0	0
11H20:	1	0	1	1	1	1	1	1	1	0
11H25:	1	0	2	1	1	1	1	1	1	1
11H30:	1	2	2	2	2	2	1	1	1	1
11H35:	1	0	1	1	1	1	1	1	1	0
11H40:	2	1	2	2	2	1	2	1	1	2
11H45:	1	0	1	1	1	1	1	0	1	1

FIN

```
***Guichet numero:  1***
   Heure de fermeture du guichet: 11H45.52mins
   Taux d'occupation de l'employe:  73.5%
   Temps moyen pour traiter une transaction: 24.3secs
   Nombre de clients servis:   250
   Attente maximum:   4.45mins
   Attente moyenne:    .99mins
   Longueur maximum de la queue:   3
   Longueur moyenne de la queue:    .96
***FIN***

***Guichet numero:  2***
   Heure de fermeture du guichet: 11H45.00mins
   Taux d'occupation de l'employe:  65.3%
   Temps moyen pour traiter une transaction: 30.0secs
   Nombre de clients servis:   182
   Attente maximum:   4.65mins
   Attente moyenne:   1.16mins
   Longueur maximum de la queue:   3
   Longueur moyenne de la queue:    .83
***FIN***

***Guichet numero:  3***
   Heure de fermeture du guichet: 11H46.57mins
   Taux d'occupation de l'employe:  97.9%
   Temps moyen pour traiter une transaction: 19.8secs
   Nombre de clients servis:    89
   Attente maximum:  24.39mins
   Attente moyenne:   5.05mins
   Longueur maximum de la queue:   4
   Longueur moyenne de la queue:   1.75
***FIN***

***Guichet numero:  4***
   Heure de fermeture du guichet: 11H45.74mins
   Taux d'occupation de l'employe:  94.9%
   Temps moyen pour traiter une transaction: 16.5secs
   Nombre de clients servis:   128
   Attente maximum:  11.34mins
   Attente moyenne:   2.99mins
   Longueur maximum de la queue:   4
   Longueur moyenne de la queue:   1.50
***FIN***

***Guichet numero:  5***
   Heure de fermeture du guichet: 11H46.93mins
   Taux d'occupation de l'employe:  87.0%
   Temps moyen pour traiter une transaction:  9.3secs
   Nombre de clients servis:   200
   Attente maximum:   5.90mins
   Attente moyenne:   1.56mins
   Longueur maximum de la queue:   3
   Longueur moyenne de la queue:   1.22
***FIN***

***Guichet numero:  6***
   Heure de fermeture du guichet: 11H45.36mins
   Taux d'occupation de l'employe:  91.0%
   Temps moyen pour traiter une transaction: 20.1secs
   Nombre de clients servis:   109
```

```
    Attente maximum:   8.98mins
    Attente moyenne:   2.89mins
    Longueur maximum de la queue:   3
    Longueur moyenne de la queue:  1.23
***FIN***

***Guichet numero:  7***
    Heure de fermeture du guichet: 11H45.80mins
    Taux d'occupation de l'employe:  87.7%
    Temps moyen pour traiter une transaction: 16.8secs
    Nombre de clients servis:   107
    Attente maximum:  14.12mins
    Attente moyenne:   2.78mins
    Longueur maximum de la queue:   3
    Longueur moyenne de la queue:  1.16
***FIN***

***Guichet numero:  8***
    Heure de fermeture du guichet: 11H45.00mins
    Taux d'occupation de l'employe:  72.7%
    Temps moyen pour traiter une transaction: 10.4secs
    Nombre de clients servis:   174
    Attente maximum:   6.25mins
    Attente moyenne:   1.28mins
    Longueur maximum de la queue:   3
    Longueur moyenne de la queue:   .87
***FIN***

***Guichet numero:  9***
    Heure de fermeture du guichet: 11H55.01mins
    Taux d'occupation de l'employe:  84.6%
    Temps moyen pour traiter une transaction: 21.1secs
    Nombre de clients servis:    84
    Attente maximum:  13.13mins
    Attente moyenne:   3.22mins
    Longueur maximum de la queue:   3
    Longueur moyenne de la queue:  1.02
***FIN***

***Guichet numero: 10***
    Heure de fermeture du guichet: 11H46.19mins
    Taux d'occupation de l'employe:  59.1%
    Temps moyen pour traiter une transaction:  8.5secs
    Nombre de clients servis:   157
    Attente maximum:   5.14mins
    Attente moyenne:   1.15mins
    Longueur maximum de la queue:   3
    Longueur moyenne de la queue:   .70
***FIN***

<<<<<FIN DE LA SIMULATION>>>>>
```

Les paramètres du système ont été choisis de manière que le système soit proche de la saturation. Le plus souvent, l'exécution de ce programme fait apparaître une lente augmentation des files d'attente à tous les guichets sauf les guichets express; ces derniers ne présentent que rarement un phénomène de saturation : le plus souvent, ils arrivent à absorber un afflux momentané de clients.

Dans ce fascicule, figurent deux cas d'exécution relativement extrêmes. Le premier fait apparaître un fort engorgement des guichets, sauf pour les guichets express qui jouent leur rôle d'assurer une réponse rapide aux clients qui n'ont qu'un petit nombre de transactions à accomplir. Dans le second, il n'y a pas de saturation; pour tous les guichets, les queues restent constamment courtes.

Dans le premier cas, on remarque que sauf aux deux guichets express et au guichet *6*, les employés étaient plus lents que la moyenne; de plus, le nombre *1560* de clients a dépassé le nombre moyen attendu *1530*.

Le deuxième cas présente des circonstances opposées. Le nombre total *1480* est inférieur au nombre attendu. Parmi les guichets "ordinaires", seuls les guichets *6* et *9* étaient desservis par des employés moins efficaces que la moyenne. Plusieurs guichets, notamment les guichets *5, 8* et *10* avaient au contraire des employés beaucoup plus efficaces que la moyenne. Bien que cette fois-ci, les employés aux guichets express étaient peu efficaces, surtout au guichet *2*, ceci n'a pas porté à conséquence vu la situation fluide des autres guichets.

D'autres exécutions de ce programme présentent, en général, un phénomène de saturation analogue à celui du premier exemple mais d'une amplitude moindre. Par exemple, à l'heure de fermeture du bureau, on pourrait avoir des files d'attente de l'ordre de dix à quinze clients.

Il est parfois possible de traiter, au moyen d'un algorithme de simulation discrète, une application de nature déterministe. Un exemple classique est celui de la recherche du plus court chemin dans un graphe. On considère un graphe, représentant (par exemple) un réseau de routes. A chaque arête de ce graphe est associé la longueur du tronçon de route correspondant. On donne deux sommets de ce graphe; il s'agit de déterminer le plus court chemin reliant ces deux sommets.

Le modèle que l'on simulera est le suivant : Au temps zéro, on fait partir de la localité (du sommet) origine du parcours un courrier dans chacun des tronçons de route qui en sont issus. Chaque courrier parcourt le tronçon correspondant à une vitesse uniforme égale à un. Lorsqu'un courrier atteint la localité à l'autre extrémité du tronçon, on a l'une des possibilités suivantes :

- La localité a déjà été atteinte par un autre courrier. Aucune autre action n'est requise.

- La localité n'a pas encore été atteinte : à cause de la vitesse unité du courrier, l'heure à laquelle ce dernier atteint la localité est égale à la longueur du plus court chemin qui le relie à l'origine du parcours. On marque que la localité a été atteinte; il faut en particulier se rappeler de sa distance à l'origine et du tronçon de route par lequel le courrier y est parvenu.

Deux sous-cas sont alors possibles :

1. La localité considérée est la destination désirée. Dans ce cas, le plus court chemin qui y mène a été trouvé. Tenant compte des informations laissées à chaque localité intermédiaire, il est facile de mettre en évidence le chemin optimum en parcourant ce dernier à l'envers de la destination à l'origine du parcours.

2. La localité considérée n'est pas la destination désirée. Il suffit de renvoyer, depuis cette localité, un nouvel ensemble de courriers sur chacun des tronçons de route qui en sont issus et qui mènent à ces localités non encore atteintes.

Le graphe considéré étant fini, il est clair que cet algorithme s'achèvera au bout d'un temps fini. Il peut cependant se produire qu'il termine son exécution sans que la destination ne soit atteinte. Ce cas ne peut se produire que si le graphe est non connexe et, de plus, si la destination fait partie d'un sous-graphe connexe disjoint de celui dont fait partie l'origine; il n'intervient donc que si la destination est inaccessible.

Cette approche est séduisante dans le sens qu'elle se généralise facilement au cas où l'on veut chercher l'itinéraire le plus rapide entre deux points du réseau en tenant compte de circonstances telles que limitations de vitesses, embouteillages, feux de signalisation, tronçons à sens uniques, postes de douane, passages à niveaux,... . Sous cette forme généralisée, il est évident que l'on a affaire à une véritable application de simulation discrète.

Cet algorithme, sous sa forme primitive, est incorporé dans le programme *villes*; étant donné un réseau de routes et un noeud spécifique, ce programme cherche le plus court chemin depuis ce noeud à chacun des noeuds du réseau. Le listage suivant montre l'application de ce programme. Le graphe concerné comporte deux sous-réseaux connexes disjoints, l'un est situé en Suisse romande et l'autre aux Etats-Unis. On constate que les tronçons de route ont été donnés dans un ordre arbitraire; il a ensuite été cherché les plus courts chemins depuis *Lausanne, Genève* et *New York*. Pour chaque noeud du réseau, il est indiqué (pour autant qu'il soit accessible, sa distance à l'origine du parcours, puis (le cas échéant) le dernier noeud qu'il faut traverser et finalement le numéro du tronçon de route qui relie ce noeud à la destination. Ainsi, on constate que pour aller de *Lausanne* à *Pontarlier*, il faut d'abord aller à *Vallorbe* puis utiliser le tronçon de route *198* pour une distance totale de *68 km*. Remontant de proche en proche, on trouve l'itinéraire suivant : *Lausanne* - 168 - *Penthalaz (14)* - 98 - *Cossonay (16)* - 166 - *La Sarraz (21.5)* - 182 - *Croy (28)* - 172 - *Vallorbe (42)* - 198 - *Pontarlier (68)*.

```
*****Description du reseau*****
  1--->Grandsivaz Romont 18
  2--->Morges Cossonay 12
  3--->Lucens Romont 10
  4--->Annemasse Thonon 31
  5--->Massongex Saint_Maurice 3
  6--->Le_Pont Vallorbe 11
  7--->Geneve Nyon 23
  8--->Montreux Chatel_Saint_Denis 15
  9--->Geneve Douvaine 17
 10--->Allaman Aubonne 3
 11--->Morat Fribourg 17
 12--->Philadelphie New_York 160
 13--->Philadelphie Scranton 195.2
 14--->Oron Vaulruz 15
 15--->Tavernes Oron 4
 16--->Moudon Oron 12
 17--->Cottens L'Isle 8
 18--->New_York Boston 384
 19--->La_Cure Le_Brassus 17
 20--->Baltimore Harrisburg 120
 21--->Yverdon Sainte_Croix 19
 22--->Aigle Bex 9
 23--->Bex Saint_Maurice 4
 24--->Evian Saint_Gingolphe 17
 25--->Broc Jaun 19
 26--->Bulle Broc 4
 27--->Montbovon Les_Moulins 8.5
 28--->Montbovon Bulle 16.5
 29--->Chatel_Saint_Denis Vaulruz 13
 30--->Vaulruz Bulle 6
 31--->Mont_sur_Rolle Aubonne 7
 32--->Lausanne Tavernes 15
 33--->Syracuse Albany 217.6
 34--->Aubonne Biere 7
 35--->Payerne Grandcour 6
 36--->Divonne Gex 8
 37--->Neuchatel La_Chaux_de_Fonds 22
 38--->Gimel Biere 7
 39--->Morgins Monthey 15
 40--->Saint_Gingolphe Chessel 9
 41--->Villars Salavaux 3
 42--->Avenches Salavaux 6
 43--->Thonon Evian 9
 44--->Vuarrens Orbe 12
 45--->Chessel Monthey 13.5
 46--->Bulle Riaz 3
 47--->Fribourg Le_Bry 15
 48--->Romont Fribourg 24
 49--->Echallens Vuarrens 5.5
 50--->Oron Romont 17
 51--->Prahins Yverdon 13
 52--->Harrisburg Buffalo 444.8
 53--->Washington Pittsburg 353.6
 54--->Pontarlier Morteau 31
 55--->New_York Albany 246.4
 56--->Morteau Le_Locle 15
 57--->Les_Moulins Chateau_d'Oex 2
 58--->Mollendruz Croy 15
 59--->Bettens Orbe 14.5
```

```
 60--->La_Sarraz Orbe 8.5
 61--->Avenches Fribourg 15
 62--->Salavaux Sugiez 9
 63--->Cottens Cossonay 7
 64--->Allaman Morges 9
 65--->Ins Kerzers 8
 66--->Yverdon Neuchatel 38
 67--->Grandcour Villars 7
 68--->Aubonne Cottens 13
 69--->Cottens Biere 11
 70--->Morat Sugiez 6
 71--->Le_Sepey Les_Mosses 8
 72--->Grandsivaz Fribourg 12
 73--->Les_Mosses Les_Moulins 14
 74--->Les_Mosses Chateau_d'Oex 15
 75--->Lucens Payerne 16.5
 76--->Payerne Romont 17.5
 77--->Romont Vaulruz 12
 78--->Bettens Echallens 11
 79--->Carrouge Moudon 6.5
 80--->Le_Sepey Leysin 7
 81--->Massongex Bex 2
 82--->Albany Boston 260.8
 83--->Cheseaux Echallens 7
 84--->Marchairuz Biere 11
 85--->Col_du_Pillon Gstaad 17.5
 86--->Gstaad Saanen 3.5
 87--->Payerne Avenches 11
 88--->La_Chaux_de_Fonds Le_Locle 9
 89--->Rolle Allaman 5
 90--->Moudon Lucens 5.5
 91--->Le_Brassus Le_Pont 13
 92--->Pittsburg Cleveland 206.4
 93--->Rolle Mont_sur_Rolle 1
 94--->Harrisburg Philadelphie 192
 95--->Nyon Rolle 12
 96--->Pittsburg Harrisburg 302.4
 97--->Fontany Abondance 10
 98--->Penthalaz Cossonay 2
 99--->Penthalaz Bettens 6
100--->Nyon Saint_Georges 22
101--->Saint_Georges Gimel 5
102--->Payerne Grandsivaz 9
103--->Villars Avenches 4
104--->Chateau_d'Oex Saanen 12
105--->Montreux Villeneuve 5
106--->Ollon Villars_sur_Ollon 10
107--->Villars_sur_Ollon Col_de_la_Croix 8.5
108--->Le_Sepey Diablerets 8.5
109--->Diablerets Col_du_Pillon 5.5
110--->Cleveland Buffalo 309.2
111--->Saint_Georges Marchairuz 7
112--->Avenches Morat 8
113--->Estavayer Grandcour 7
114--->Harrisburg Cleveland 516.8
115--->Pittsburg Buffalo 345.6
116--->Abondance Chatel 11.5
117--->Morges Lausanne 11
118--->Lausanne Penthalaz 14
119--->Neuchatel Le_Locle 29
```

```
120--->Harrisburg Scranton 188.8
121--->Washington Baltimore 59.2
122--->Fleurier Le_Locle 28
123--->Estavayer Payerne 10
124--->Saanen Zweisimmen 14
125--->Zweisimmen Boltigen 10
126--->Jaunpass Boltigen 10
127--->Jaun Jaunpass 7
128--->Vevey Montreux 7
129--->Lausanne Possens 29
130--->Morges Cottens 9
131--->Sainte_Croix Fleurier 13
132--->Pontarlier Fleurier 21
133--->Orbe Vallorbe 18
134--->Fribourg Le_Mouret 9
135--->Romont Le_Bry 16
136--->Le_Bry Riaz 9
137--->Riaz La_Roche 9
138--->Sugiez Ins 5
139--->Scranton Syracuse 217.6
140--->Vuarrens Yverdon 11
141--->Sainte_Croix Pontarlier 20
142--->Tavernes Vevey 16
143--->Nyon Divonne 10
144--->Baltimore Philadelphie 153.6
145--->Mont_sur_Rolle Gimel 8
146--->Mollendruz Le_Pont 2.5
147--->Vevey Chatel_Saint_Denis 12
148--->Lausanne Cheseaux 8.5
149--->Cheseaux Bettens 6
150--->Carrouge Tavernes 6.5
151--->Buffalo Syracuse 235.2
152--->Croy Orbe 6
153--->Biere L'Isle 12
154--->Lausanne Vevey 18
155--->Rennaz Aigle 7
156--->L'Isle Mollendruz 9.5
157--->Scranton Buffalo 393.6
158--->New_York Scranton 220.8
159--->Scranton Albany 276.8
160--->Cossonay L'Isle 9
161--->Oron Chatel_Saint_Denis 10.5
162--->Col_de_la_Croix Diablerets 8.5
163--->Thierrens Prahins 4
164--->Monthey Massongex 3.5
165--->Douvaine Thonon 16
166--->Cossonay La_Sarraz 5.5
167--->Nyon La_Cure 22
168--->Geneve Gex 17
169--->Prahins Estavayer 17
170--->Bex Villars_sur_Ollon 15
171--->Aigle Ollon 5
172--->Croy Vallorbe 14
173--->Chessel Rennaz 4
174--->Villeneuve Rennaz 3
175--->Thonon Fontany 17.5
176--->Marchairuz Le_Brassus 7
177--->Morat Kerzers 9
178--->Fleurier Neuchatel 31
179--->Aigle Le_Sepey 10
```

```
180--->Echallens Possens 7.5
181--->Gex La_Cure 14
182--->La_Sarraz Croy 6.5
183--->Lausanne Carrouge 17.5
184--->Evian Fontany 18
185--->Orbe Yverdon 13
186--->Le_Mouret La_Roche 6
187--->Possens Moudon 10
188--->Neuchatel Ins 15
189--->Yverdon Estavayer 19
190--->Washington Harrisburg 171.2
191--->Aubonne Gimel 7
192--->Chatel Morgins 4.5
193--->Nyon Mont_sur_Rolle 13
194--->Geneve Annemasse 7
195--->Echallens Prahins 16
196--->Possens Thierrens 7
197--->Moudon Thierrens 7
198--->Vallorbe Pontarlier 26
<<<FIN>>>
```

La figure 50 montre le réseau concerné; les itinéraires choisis par le programme depuis *Lausanne* et *New York* y sont indiqués en traits gras. Il va de soi que ce sous-graphe forme un arbre dont la racine est la localité depuis laquelle ont été calculés les itinéraires minimaux.

```
***Itineraires depuis Lausanne***

    Pontarlier              68.0Km. via Vallorbe et route 198
    Cleveland             ---localite inaccessible---
    Penthalaz               14.0Km. via route 118
    Divonne                 47.0Km. via Nyon et route 143
    Nyon                    37.0Km. via Rolle et route  95
    Baltimore             ---localite inaccessible---
    Boltigen                80.0Km. via Jaunpass et route 126
    Payerne                 46.0Km. via Lucens et route  75
    Le Bry                  52.0Km. via Riaz et route 136
    Vevey                   18.0Km. via route 154
    Tavernes                15.0Km. via route  32
    Marchairuz              41.0Km. via Biere et route  84
    Fleurier                64.0Km. via Sainte Croix et route 131
    Villars                 59.0Km. via Grandcour et route  67
    Thonon                  72.0Km. via Evian et route  43
    Grandcour               52.0Km. via Payerne et route  35
    Boston                ---localite inaccessible---
    Zweisimmen              90.0Km. via Boltigen et route 125
    Biere                   30.0Km. via Aubonne et route  34
    La Roche                52.0Km. via Riaz et route 137
    Riaz                    43.0Km. via Bulle et route  46
    Le Locle                92.0Km. via Fleurier et route 122
    Broc                    44.0Km. via Bulle et route  26
    Cottens                 20.0Km. via Morges et route 130
    Villars sur Ollon       55.0Km. via Ollon et route 106
    Lausanne              ***origine du parcours***
    Harrisburg            ---localite inaccessible---
    Morteau                 99.0Km. via Pontarlier et route  54
    Aubonne                 23.0Km. via Allaman et route  10
    Estavayer               48.5Km. via Prahins et route 169
    Morgins                 65.5Km. via Monthey et route  39
    Pittsburg             ---localite inaccessible---
    Les Mosses              58.0Km. via Le Sepey et route  71
    Abondance               81.5Km. via Chatel et route 116
    Villeneuve              30.0Km. via Montreux et route 105
    Avenches                57.0Km. via Payerne et route  87
    Grandsivaz              54.0Km. via Romont et route   1
    New York              ---localite inaccessible---
    Le Brassus              48.0Km. via Marchairuz et route 176
    Chessel                 37.0Km. via Rennaz et route 173
    Mont sur Rolle          26.0Km. via Rolle et route  93
    La Cure                 59.0Km. via Nyon et route 167
    Les Moulins             65.0Km. via Montbovon et route  27
    Croy                    28.0Km. via La Sarraz et route 182
    La Chaux de Fonds       92.0Km. via Neuchatel et route  37
    Allaman                 20.0Km. via Morges et route  64
    Annemasse               67.0Km. via Geneve et route 194
    Diablerets              58.5Km. via Le Sepey et route 108
    Romont                  36.0Km. via Oron et route  50
    Le Mouret               58.0Km. via La Roche et route 186
    Saint Gingolphe         46.0Km. via Chessel et route  40
    Moudon                  24.0Km. via Carrouge et route  79
    Kerzers                 74.0Km. via Morat et route 177
    Rennaz                  33.0Km. via Villeneuve et route 174
    Chatel Saint Denis      29.5Km. via Oron et route 161
    Bulle                   40.0Km. via Vaulruz et route  30
    Monthey                 50.5Km. via Chessel et route  45
    Jaun                    63.0Km. via Broc et route  25
```

Jaunpass	70.0Km. via Jaun et route 127
Montreux	25.0Km. via Vevey et route 128
Gstaad	81.5Km. via Col du Pillon et route 85
Mollendruz	34.5Km. via L'Isle et route 156
Vallorbe	42.0Km. via Croy et route 172
Echallens	15.5Km. via Cheseaux et route 83
Fontany	81.0Km. via Evian et route 184
Gimel	30.0Km. via Aubonne et route 191
Sainte Croix	51.0Km. via Yverdon et route 21
Rolle	25.0Km. via Allaman et route 89
Syracuse	---localite inaccessible---
Chatel	70.0Km. via Morgins et route 192
Montbovon	56.5Km. via Bulle et route 28
Le Pont	37.0Km. via Mollendruz et route 146
Gex	55.0Km. via Divonne et route 36
Orbe	29.0Km. via Bettens et route 59
Cossonay	16.0Km. via Penthalaz et route 98
Bettens	14.5Km. via Cheseaux et route 149
Le Sepey	50.0Km. via Aigle et route 179
Possens	23.0Km. via Echallens et route 180
Yverdon	32.0Km. via Vuarrens et route 140
Saanen	79.0Km. via Chateau d'Oex et route 104
Prahins	31.5Km. via Echallens et route 195
Philadelphie	---localite inaccessible---
Bex	49.0Km. via Aigle et route 22
Saint Georges	35.0Km. via Gimel et route 101
Morges	11.0Km. via route 117
Sugiez	71.0Km. via Salavaux et route 62
Neuchatel	70.0Km. via Yverdon et route 66
Oron	19.0Km. via Tavernes et route 15
Morat	65.0Km. via Avenches et route 112
L'Isle	25.0Km. via Cossonay et route 160
Geneve	60.0Km. via Nyon et route 7
Buffalo	---localite inaccessible---
Evian	63.0Km. via Saint Gingolphe et route 24
Saint Maurice	53.0Km. via Bex et route 23
Vuarrens	21.0Km. via Echallens et route 49
Col du Pillon	64.0Km. via Diablerets et route 109
Washington	---localite inaccessible---
Ins	76.0Km. via Sugiez et route 138
Cheseaux	8.5Km. via route 148
Aigle	40.0Km. via Rennaz et route 155
Carrouge	17.5Km. via route 183
Lucens	29.5Km. via Moudon et route 90
Vaulruz	34.0Km. via Oron et route 14
Scranton	---localite inaccessible---
Massongex	51.0Km. via Bex et route 81
Salavaux	62.0Km. via Villars et route 41
Ollon	45.0Km. via Aigle et route 171
Albany	---localite inaccessible---
Leysin	57.0Km. via Le Sepey et route 80
Douvaine	77.0Km. via Geneve et route 9
Thierrens	30.0Km. via Possens et route 196
Chateau d'Oex	67.0Km. via Les Moulins et route 57
Fribourg	60.0Km. via Romont et route 48
Col de la Croix	63.5Km. via Villars sur Ollon et route 107
La Sarraz	21.5Km. via Cossonay et route 166

<<<FIN>>>

```
***Itineraires depuis Geneve***
    Pontarlier           98.0Km. via Vallorbe et route 198
    Cleveland            ---localite inaccessible---
    Penthalaz            63.0Km. via Cossonay et route  98
    Divonne              25.0Km. via Gex et route  36
    Nyon                 23.0Km. via route   7
    Baltimore            ---localite inaccessible---
    Boltigen            140.0Km. via Jaunpass et route 126
    Payerne             106.0Km. via Lucens et route  75
    Le Bry              112.0Km. via Riaz et route 136
    Vevey                78.0Km. via Lausanne et route 154
    Tavernes             75.0Km. via Lausanne et route  32
    Marchairuz           52.0Km. via Saint Georges et route 111
    Fleurier            119.0Km. via Pontarlier et route 132
    Villars             119.0Km. via Grandcour et route  67
    Thonon               33.0Km. via Douvaine et route 165
    Grandcour           112.0Km. via Payerne et route  35
    Boston               ---localite inaccessible---
    Zweisimmen          138.0Km. via Saanen et route 124
    Biere                50.0Km. via Aubonne et route  34
    La Roche            112.0Km. via Riaz et route 137
    Riaz                103.0Km. via Bulle et route  46
    Le Locle            144.0Km. via Morteau et route  56
    Broc                104.0Km. via Bulle et route  26
    Cottens              56.0Km. via Aubonne et route  68
    Villars sur Ollon    94.0Km. via Ollon et route 106
    Lausanne             60.0Km. via Morges et route 117
    Harrisburg           ---localite inaccessible---
    Morteau             129.0Km. via Pontarlier et route  54
    Aubonne              43.0Km. via Mont sur Rolle et route  31
    Estavayer           107.0Km. via Yverdon et route 189
    Morgins              76.5Km. via Chatel et route 192
    Pittsburg            ---localite inaccessible---
    Les Mosses           97.0Km. via Le Sepey et route  71
    Abondance            60.5Km. via Fontany et route  97
    Villeneuve           75.0Km. via Rennaz et route 174
    Avenches            117.0Km. via Payerne et route  87
    Grandsivaz          114.0Km. via Romont et route   1
    New York             ---localite inaccessible---
    Le Brassus           48.0Km. via La Cure et route  19
    Chessel              68.0Km. via Saint Gingolphe et route  40
    Mont sur Rolle       36.0Km. via Nyon et route 193
    La Cure              31.0Km. via Gex et route 181
    Les Moulins         111.0Km. via Les Mosses et route  73
    Croy                 73.0Km. via La Sarraz et route 182
    La Chaux de Fonds   148.0Km. via Neuchatel et route  37
    Allaman              40.0Km. via Rolle et route  89
    Annemasse             7.0Km. via route 194
    Diablerets           97.5Km. via Le Sepey et route 108
    Romont               96.0Km. via Oron et route  50
    Le Mouret           118.0Km. via La Roche et route 186
    Saint Gingolphe      59.0Km. via Evian et route  24
    Moudon               84.0Km. via Carrouge et route  79
    Kerzers             134.0Km. via Morat et route 177
    Rennaz               72.0Km. via Chessel et route 173
    Chatel Saint Denis   89.5Km. via Oron et route 161
    Bulle               100.0Km. via Vaulruz et route  30
    Monthey              81.5Km. via Chessel et route  45
    Jaun                123.0Km. via Broc et route  25
    Jaunpass            130.0Km. via Jaun et route 127
```

```
Montreux            80.0Km. via Villeneuve et route 105
Gstaad             120.5Km. via Col du Pillon et route  85
Mollendruz          63.5Km. via Le Pont et route 146
Vallorbe            72.0Km. via Le Pont et route   6
Echallens           75.5Km. via Cheseaux et route  83
Fontany             50.5Km. via Thonon et route 175
Gimel               44.0Km. via Mont sur Rolle et route 145
Sainte Croix       107.0Km. via Yverdon et route  21
Rolle               35.0Km. via Nyon et route  95
Syracuse          ---localite inaccessible---
Chatel              72.0Km. via Abondance et route 116
Montbovon          116.5Km. via Bulle et route  28
Le Pont             61.0Km. via Le Brassus et route  91
Gex                 17.0Km. via route 168
Orbe                75.0Km. via La Sarraz et route  60
Cossonay            61.0Km. via Morges et route   2
Bettens             69.0Km. via Penthalaz et route  99
Le Sepey            89.0Km. via Aigle et route 179
Possens             83.0Km. via Echallens et route 180
Yverdon             88.0Km. via Orbe et route 185
Saanen             124.0Km. via Gstaad et route  86
Prahins             91.5Km. via Echallens et route 195
Philadelphie      ---localite inaccessible---
Bex                 87.0Km. via Massongex et route  81
Saint Georges       45.0Km. via Nyon et route 100
Morges              49.0Km. via Allaman et route  64
Sugiez             131.0Km. via Salavaux et route  62
Neuchatel          126.0Km. via Yverdon et route  66
Oron                79.0Km. via Tavernes et route  15
Morat              125.0Km. via Avenches et route 112
L'Isle              62.0Km. via Biere et route 153
Geneve            ***origine du parcours***
Buffalo           ---localite inaccessible---
Evian               42.0Km. via Thonon et route  43
Saint Maurice       88.0Km. via Massongex et route   5
Vuarrens            81.0Km. via Echallens et route  49
Col du Pillon      103.0Km. via Diablerets et route 109
Washington        ---localite inaccessible---
Ins                136.0Km. via Sugiez et route 138
Cheseaux            68.5Km. via Lausanne et route 148
Aigle               79.0Km. via Rennaz et route 155
Carrouge            77.5Km. via Lausanne et route 183
Lucens              89.5Km. via Moudon et route  90
Vaulruz             94.0Km. via Oron et route  14
Scranton          ---localite inaccessible---
Massongex           85.0Km. via Monthey et route 164
Salavaux           122.0Km. via Villars et route  41
Ollon               84.0Km. via Aigle et route 171
Albany            ---localite inaccessible---
Leysin              96.0Km. via Le Sepey et route  80
Douvaine            17.0Km. via route   9
Thierrens           90.0Km. via Possens et route 196
Chateau d'Oex      112.0Km. via Les Mosses et route  74
Fribourg           120.0Km. via Romont et route  48
Col de la Croix    102.5Km. via Villars sur Ollon et route 107
La Sarraz           66.5Km. via Cossonay et route 166
<<<FIN>>>
```

Itineraires depuis New York

Pontarlier	---localite inaccessible---
Cleveland	860.8Km. via Pittsburg et route 92
Penthalaz	---localite inaccessible---
Divonne	---localite inaccessible---
Nyon	---localite inaccessible---
Baltimore	313.6Km. via Philadelphie et route 144
Boltigen	---localite inaccessible---
Payerne	---localite inaccessible---
Le Bry	---localite inaccessible---
Vevey	---localite inaccessible---
Tavernes	---localite inaccessible---
Marchairuz	---localite inaccessible---
Fleurier	---localite inaccessible---
Villars	---localite inaccessible---
Thonon	---localite inaccessible---
Grandcour	---localite inaccessible---
Boston	384.0Km. via route 18
Zweisimmen	---localite inaccessible---
Biere	---localite inaccessible---
La Roche	---localite inaccessible---
Riaz	---localite inaccessible---
Le Locle	---localite inaccessible---
Broc	---localite inaccessible---
Cottens	---localite inaccessible---
Villars sur Ollon	---localite inaccessible---
Lausanne	---localite inaccessible---
Harrisburg	352.0Km. via Philadelphie et route 94
Morteau	---localite inaccessible---
Aubonne	---localite inaccessible---
Estavayer	---localite inaccessible---
Morgins	---localite inaccessible---
Pittsburg	654.4Km. via Harrisburg et route 96
Les Mosses	---localite inaccessible---
Abondance	---localite inaccessible---
Villeneuve	---localite inaccessible---
Avenches	---localite inaccessible---
Grandsivaz	---localite inaccessible---
New York	***origine du parcours***
Le Brassus	---localite inaccessible---
Chessel	---localite inaccessible---
Mont sur Rolle	---localite inaccessible---
La Cure	---localite inaccessible---
Les Moulins	---localite inaccessible---
Croy	---localite inaccessible---
La Chaux de Fonds	---localite inaccessible---
Allaman	---localite inaccessible---
Annemasse	---localite inaccessible---
Diablerets	---localite inaccessible---
Romont	---localite inaccessible---
Le Mouret	---localite inaccessible---
Saint Gingolphe	---localite inaccessible---
Moudon	---localite inaccessible---
Kerzers	---localite inaccessible---
Rennaz	---localite inaccessible---
Chatel Saint Denis	---localite inaccessible---
Bulle	---localite inaccessible---
Monthey	---localite inaccessible---
Jaun	---localite inaccessible---
Jaunpass	---localite inaccessible---
Montreux	---localite inaccessible---

```
    Gstaad                     ---localite inaccessible---
    Mollendruz                 ---localite inaccessible---
    Vallorbe                   ---localite inaccessible---
    Echallens                  ---localite inaccessible---
    Fontany                    ---localite inaccessible---
    Gimel                      ---localite inaccessible---
    Sainte Croix               ---localite inaccessible---
    Rolle                      ---localite inaccessible---
    Syracuse                    438.4Km. via Scranton et route 139
    Chatel                     ---localite inaccessible---
    Montbovon                  ---localite inaccessible---
    Le Pont                    ---localite inaccessible---
    Gex                        ---localite inaccessible---
    Orbe                       ---localite inaccessible---
    Cossonay                   ---localite inaccessible---
    Bettens                    ---localite inaccessible---
    Le Sepey                   ---localite inaccessible---
    Possens                    ---localite inaccessible---
    Yverdon                    ---localite inaccessible---
    Saanen                     ---localite inaccessible---
    Prahins                    ---localite inaccessible---
    Philadelphie                160.0Km. via route  12
    Bex                        ---localite inaccessible---
    Saint Georges              ---localite inaccessible---
    Morges                     ---localite inaccessible---
    Sugiez                     ---localite inaccessible---
    Neuchatel                  ---localite inaccessible---
    Oron                       ---localite inaccessible---
    Morat                      ---localite inaccessible---
    L'Isle                     ---localite inaccessible---
    Geneve                     ---localite inaccessible---
    Buffalo                     614.4Km. via Scranton et route 157
    Evian                      ---localite inaccessible---
    Saint Maurice              ---localite inaccessible---
    Vuarrens                   ---localite inaccessible---
    Col du Pillon              ---localite inaccessible---
    Washington                  372.8Km. via Baltimore et route 121
    Ins                        ---localite inaccessible---
    Cheseaux                   ---localite inaccessible---
    Aigle                      ---localite inaccessible---
    Carrouge                   ---localite inaccessible---
    Lucens                     ---localite inaccessible---
    Vaulruz                    ---localite inaccessible---
    Scranton                    220.8Km. via route 158
    Massongex                  ---localite inaccessible---
    Salavaux                   ---localite inaccessible---
    Ollon                      ---localite inaccessible---
    Albany                      246.4Km. via route  55
    Leysin                     ---localite inaccessible---
    Douvaine                   ---localite inaccessible---
    Thierrens                  ---localite inaccessible---
    Chateau d'Oex              ---localite inaccessible---
    Fribourg                   ---localite inaccessible---
    Col de la Croix            ---localite inaccessible---
    La Sarraz                  ---localite inaccessible---
<<<FIN>>>

<<<<<FIN DES APPLICATIONS>>>>>
```

Le programme *villes* a la structure donnée à la figure 51.

program *villes*

integer **subrange** *naturel* **subrange** *positif*

coroutine *échéancier*

real **variable** *heure* **value** *horloge*
procedure *attendre (real* **value** *durée)*

class *ville (string* **value** *nom)*

ville **function** *relier*
(positif **value** *numéro_route;real* **value** *distance;*
ville **value** *destination)*
Boolean **variable** *att* **value** *atteinte*
ville **variable** *etp* **value** *étape*
naturel **variable** *itin* **value** *itinéraire*
real **variable** *dist* **value** *distance*
process *courrier*
(ville **value** *origine; naturel* **value** *numéro;*
real **value** *longueur)*
ville **expression** *init*
ville **expression** *origine*

module *réseau* **index** *localité*

ville **function** *localité (string* **value** *nom)*
Boolean **function** *existe (string* **value** *nom)*
actor *action (ville* **value** *localité)*
procedure *pour_chaque_ville (action* **value** *acte)*

string **variable** *nom_de_ville*

Fig. 51

```
/* /*OLDSOURCE=USER2:[RAPIN]VILLES.NEW*/ */
PROGRAM villes DECLARE

  integer SUBRANGE naturel(naturel>=0)
          SUBRANGE positif(positif>0);

COROUTINE echeancier
  ATTRIBUTE horloge,attendre
(*Un echeancier pour les applications de la simulation discrete*)
DECLARE(*echeancier*)
  real VARIABLE heure VALUE horloge;
  (*L'heure courante*)

  OBJECT notice_arbre(activity VALUE acte; real VALUE temps;
                      notice_arbre VARIABLE gauche,droite)
    VARIABLE racine:=NIL;

  notice_arbre FUNCTION union
    (notice_arbre VARIABLE p,q)
  (*Fusionne les arbres de priorite  p  et  q ; le resultat est
    l'arbre fusionne.
  *)
  DECLARE notice_arbre REFERENCE r->p DO
    WITHIN q REPEAT
      IF TAKE
        UNLESS r=NIL THEN
          (*q.*)temps<r.temps
        DEFAULT TRUE DONE
      THEN r:=:q DONE;
      (* r designe la notice prioritaire*)
      CONNECT r THEN
        r->gauche:=:droite
      DONE
    REPETITION
  TAKE p DONE(*union*);

  PROCEDURE attendre(real VALUE duree)DO
  (*Le preocessus CURRENT est suspendu pendant la duree donnee; sans
   effet si duree<0
  *)
    UNLESS duree<0 THEN
      racine:=union(racine,notice_arbre(CURRENT,horloge+duree,NIL,NIL))
    RETURN DONE
  DONE(*attendre*)
DO(*echeancier*)LOOP
  heure:=0
RETURN
  WITHIN racine REPEAT
    (*Enleve de l'arbre de priorite la prochaine notice d'evenement
      a prendre en compte, avance l'horloge et effectue l'action
      correspondante.
    *)
    heure:=temps; racine:=union(gauche,droite);
    ACTIVATE acte NOW
  REPETITION
  (*La simulation est terminee; en prepare une autre*)
REPETITION(*echeancier*)DONE;
/* /*EJECT*/ */
```

L'échéancier est une version modifiée de celui qui figure dans le programme *guichets*. Seule la primitive *attendre* a été nécessaire. D'autre part, cet échéancier permet de faire plusieurs

simulations lors d'une seule exécution du programme. Sa partie exécutable est encapsulée dans un énoncé **loop**; ainsi, une fois une simulation terminée, l'horloge est remise à zéro et l'échéancier se détache, prêt à faire une nouvelle simulation lorsqu'on le réactive.

Au moyen de la liste d'objets *tête* du type *tronçon*, un objet du type *ville* contient la description des tronçons de route qui y aboutissent; ceux-ci sont créés par l'application de la procédure attribut *relier*. L'algorithme des courriers est incorporé, sous la forme du type processus *courrier*, dans la classe *ville*; cet algorithme modélise un courrier qui aboutit, par un tronçon de route donné, à la ville concernée. Après s'être assuré que la ville n'a pas été atteinte par un autre courrier, il met à jour les variables décrivant l'itinéraire parcouru et lance un nouvel ensemble de courriers vers les destinations accessibles depuis la localité concernée au moyen d'un seul tronçon. On remarque que le type processus *courrier* n'est pas exporté comme attribut du type *ville*. Pour obtenir les chemins minimaux depuis une localité donnée, il suffit d'appliquer à cette dernière la primitive *origine* après s'être assuré, au moyen de la fonction *init*, que toutes les localités du réseau sont considérées comme non atteintes. La fonction *origine* a été implantée en faisant arriver un courrier bidon à la localité concernée; ceci placera une notice d'événement dans l'échéancier (à cause du *attendre* au début de l'algorithme du courrier). Il suffit ensuite de lancer la simulation en réactivant l'échéancier : l'algorithme du courrier bidon impliquera le lancement des courriers initiaux issus de la localité origine.

Dans le module *réseau*, il est enregistré le graphe sur lequel portera l'application. Il y est construit une table extensible *les_villes* des noeuds du réseau. Au moyen de cette table et de la fonction d'indiçage exportée du module, il est possible d'atteindre une localité quelconque donnée par son nom; si la localité ne figure pas dans la table, cette opération d'indiçage crée l'objet correspondant et le stocke dans la table. Il est également exporté de ce module l'itérateur *pour_chaque_ville* qui permet de faire une action donnée *acte* pour chacune des localités du réseau.

```
CLASS ville
  VALUE moi
  ATTRIBUTE nom,relier,atteinte,etape,itineraire,distance,
            init,origine
  (string VALUE nom)
DECLARE(*ville*)
  OBJECT troncon
    (positif VALUE numero; real VALUE longueur;
     ville VALUE voisin; troncon VALUE suivant)
    VARIABLE tete:=NIL;

  ville FUNCTION relier
    (positif VALUE numero_route; real VALUE distance;
     ville VALUE destination)
  (*Relie la ville concernee a la ville  destination au moyen
    du troncon de route  numero_route  de longueur  distance .
  *)
  DO(*relier*)
    tete:=troncon(numero_route,distance,destination,tete)
  TAKE moi DONE(*relier*);

  Boolean VARIABLE att VALUE atteinte:=FALSE;
  (*Vrai ssi la ville concernee a ete atteinte par un courrier*)

  ville VARIABLE etp VALUE etape;
  (*La ville precedante du parcours menant a la ville concernee.

    Condition d'emploi:  atteinte
  *)

  naturel VARIABLE itin VALUE itineraire;
  (*Le troncon de route par lequel la ville concernee a ete
    atteinte.

    Condition d'emploi: atteinte
  *)

  real VARIABLE dist VALUE distance;
  (*La distance depuis la ville origine du parcours.

    Condition d'emploi:  atteinte
  *)

  PROCESS courrier
    (ville VALUE origine; naturel VALUE numero;
     real VALUE longueur)
  (*L'algorithme du courrier qui joint la ville concernee a
    partir de la ville  origine  au moyen du troncon de route
    numero  de  longueur  donnee .
  *)
  DO(*courrier*)
    (*le courrier est en marche... *)
```

```
      attendre(longueur);
       (*il a atteint la ville concernee*)
      UNLESS atteinte(*par un autre courrier*)THEN
        att:=TRUE;
        etp:=origine;
        itin:=numero;
        dist:=horloge;
         (*lance des courriers vers les villes voisines*)
        DECLARE troncon VARIABLE curs:=tete DO
          WITHIN curs REPEAT
            UNLESS voisin.atteinte THEN
             voisin.courrier(moi,(*curs.*)numero,(*curs.*)longueur)
            DONE;
             curs:=suivant
          REPETITION
        DONE(*DECLARE curs*)
      DONE(*UNLESS atteinte*)
    DONE(*courrier*);

    ville EXPRESSION init=
    (*reinitialise la ville en vue du calcul de sa distance depuis
      une ville arbitraire du reseau; le resultat est la ville
      concernee  moi .
    *)
      (att:=FALSE; moi);

    ville EXPRESSION origine=
    (*recherche le plus court chemin menant de la ville concernee
      a chacune des villes du reseau; le resultat est la ville
      concernee  moi .

      Condition d'emploi: init  a ete effectuee pour toutes les
        villes du reseau au prealable.
    *)
      ((*fait arriver un courrier bidon a la ville concernee*)
       courrier(NIL,0,0);
       (*lance la simulation*)
       ACTIVATE echeancier NOW;
       (*celle-ci est achevee: les distances requises sont
         evaluees; retourne le resultat.
       *)
      moi)
  DO(*ville*)DONE;
/* /*EJECT*/ */
```

```
393  MODULE reseau
395    INDEX localite
397    ATTRIBUTE existe,action,pour_chaque_ville
403   (*Enregistre le reseau concerne; chaque troncon de route y est
403    presente sous la forme d'une ligne de texte contenant, sous
403    la forme de chaines, le nom des deux localites encadrantes
403    suivi de sa longueur. La description du reseau sera suivie
403    d'une ligne contenant la chaine  <<<FIN>>> .
403  *)
403  DECLARE(*reseau*)
404    CONSTANT taille_init=20,rempli_min=.5,rempli_max=.8;
419
419    ville TABLE vitable VARIABLE les_villes:=vitable(taille_init);
430
430    ville FUNCTION localite
433      (string VALUE nom)
438    (*le resultat est la ville de  nom  donne; si celle-ci ne figure
438      pas dans le repertoire, la cree et l'y insere
438    *)
438    DO(*localite*)
439      UNLESS les_villes ENTRY nom THEN
444        les_villes[nom]:=ville(nom);
454        IF CARD les_villes>rempli_max*CAPACITY les_villes THEN
463          THROUGH
464            (vitable(CARD les_villes/rempli_min)=:les_villes)
475          INDEX nom  VALUE ville_nom
479          REPEAT les_villes[nom]:=ville_nom REPETITION
487        DONE
488      DONE
489    TAKE les_villes[nom] DONE(*localite*);
496
496    Boolean FUNCTION existe
499      (string VALUE nom)
504    (*vrai ssi  nom  est un nom de localite du reseau*)
504    DO TAKE les_villes ENTRY nom DONE;
511
511    ACTOR action(ville VALUE localite);
519
519    PROCEDURE pour_chaque_ville
521      (action VALUE acte)
526    (*Effectue  acte[localite]  pour chaque  localite  du reseau;
526      l'ordre de prise en compte des localites n'est pas defini
526      a priori.
526    *)
526    DO(*pour_chaque_ville*)
527      THROUGH
528        les_villes VALUE localite
531      REPEAT
532        acte[localite]
536      REPETITION
537    DONE(*pour_chaque_ville*)
538  DO(*reseau*)
```

```
   print("*****Description du reseau*****",line);
   DECLARE
     naturel VARIABLE numero:=0;
     (*Le numero du dernier troncon*)

     string VARIABLE texte;
     (*La derniere ligne lue du fichier de donnees*)

     CONSTANT alpha={FROM "A" TO "Z",FROM "a" TO "z","-","'","_"};
     (*Les caracteres susceptibles d'apparaitre dans les noms de
       ville; le caractere de soulignage y sera systematiquement
       remplace par l'espace blanc.
     *)

     string EXPRESSION nom=
     (*Extrait du debut de  texte  un nom de localite*)
       DECLARE
          string VARIABLE res:=alpha SPAN (texte:=" "-texte)
       DO
          WHILE "_" IN res REPEAT
            res:=res LEFTOF "_"+" "+res LEFTCUT "_"
          REPETITION;
          texte:=alpha-texte
       TAKE res DONE;

     CONSTANT num={FROM "0" TO "9"};

     real EXPRESSION nombre=
     (*Extrait du debut de  texte  une valeur reelle*)
       DECLARE
          string VALUE
            ent=num SPAN (texte:=" "-texte),
            frc=IF "." STARTS (texte:=num-texte) THEN
                   texte:=texte LEFTCUT "."
                 TAKE
                   num SPAN (num-texte=:texte)
                 DEFAULT "" DONE;
          real VARIABLE res:=0
       DO
          THROUGH ent+frc VALUE dig REPEAT
            res:=10*res+(ORD dig-ORD "0")
          REPETITION
       TAKE res*10**(-LENGTH frc) DONE;

     string VARIABLE org,dest; real VARIABLE long
   DO
     UNTIL
       read(texte)
     TAKE " "-texte-" "="<<<FIN>>>" REPEAT
       edit((numero:=SUCC numero),4,0);
       print("--->",texte,line);
       org:=nom; dest:=nom; long:=nombre;
```

```
          IF " "-texte="" THEN
            localite(org).relier(numero,long,localite(dest));
            localite(dest).relier(numero,long,localite(org))
          DEFAULT
            print(___"###ligne precedente incorrecte ignoree###",line)
          DONE
        REPETITION
      DONE(*DECLARE numero,... *);
      print("<<<FIN>>>",page)
    DONE(*reseau*);
    /* /*EJECT*/ */
```

Le rôle de la partie exécutable du programme *villes* est d'enregistrer, une à une, les localités depuis lesquelles les itinéraires minimaux doivent être construits et de calculer ces derniers.

```
      string VARIABLE nom_de_ville
    DO(*villes*)
      UNTIL end_file REPEAT
        read(nom_de_ville); nom_de_ville:=" "-nom_de_ville-" ";
        WHILE "_" IN nom_de_ville REPEAT
    ......nom_de_ville:=nom_de_ville LEFTOF "_"+" "+nom_de_ville LEFTCUT "_"
        REPETITION;
        print("***Itineraires depuis"_,nom_de_ville,"***",line,line);
        IF existe(nom_de_ville) THEN
          pour_chaque_ville(BODY action DO localite.init DONE);
          reseau[nom_de_ville].origine;
          pour_chaque_ville
            (BODY
               action(ville VALUE localite)
             DO
               CONNECT localite THEN
                 print(___,nom); column(25);
                 IF atteinte THEN
                   IF nom=nom_de_ville THEN
                     print("***origine du parcours***")
                   DEFAULT
                     edit(distance,6,1); print("Km. via"_);
                     UNLESS etape.nom=nom_de_ville THEN
                       print(etape.nom,_"et"_)
                     DONE;
                     print("route"_); edit(itineraire,3,0)
                   DONE
                 DEFAULT print("---localite inaccessible---") DONE
               DONE;
               line
             DONE);
          print("<<<FIN>>>",line,line)
        DEFAULT print(___,"###localite inconnue###",line) DONE
      REPETITION;
      print("<<<<<FIN DES APPLICATIONS>>>>>")
    DONE(*villes*)

    ****  No messages were issued  ****
```

Chapitre 14

Tables ordonnées extensibles

Les tables associatives gérées par une fonction de hachage, introduites au chapitre 8, représentent une structure de données versatile; on y a eu recours dans de nombreux genres d'application. Elles présentent cependant quelques inconvénients; en particulier, s'il est facile de réaliser un itérateur, ce dernier visitera les éléments de la table dans un ordre à priori arbitraire. Ainsi, les résultats du programme *villes* du chapitre précédent ne sont pas faciles à dépouiller; il aurait été préférable de lister, dans l'ordre alphabétique, les localités pour lesquelles on a établi l'itinéraire optimum. Un autre inconvénient, tout au moins dans le cas des tables prédéfinies du langage Newton gérées au moyen d'un double hachage, il est impossible de réaliser un destructeur économique : l'élimination d'un élément peut impliquer une restructuration complète de la table.

On va montrer une version améliorée du programme *villes*; cette dernière illustre deux techniques d'implantation possibles de tables, de capacité non bornée à priori, dont les entrées sont ordonnées selon l'ordre alphabétique croissant des noms des localités concernées. La première de ces techniques est très classique : il s'agit de l'arbre de recherche; l'autre en est une généralisation : elle fait appel à des directoires. On présente d'abord un exemple d'exécution de ce programme modifié.

```
*****Description du reseau*****
   1--->Grandsivaz Romont 18
   2--->Morges Cossonay 12
   3--->Lucens Romont 10
   4--->Annemasse Thonon 31
   5--->Massongex Saint_Maurice 3
   6--->Le_Pont Vallorbe 11
   7--->Geneve Nyon 23
   8--->Bern Solothurn 36
   9--->Biel Bern 33
  10--->Montreux Chatel_Saint_Denis 15
  11--->Geneve Douvaine 17
  12--->Allaman Aubonne 3
  13--->Morat Fribourg 17
  14--->Philadelphie New_York 160
  15--->Philadelphie Scranton 195.2
  16--->Oron Vaulruz 15
  17--->Tavernes Oron 4
  18--->Moudon Oron 12
  19--->Cottens L'Isle 8
  20--->New_York Boston 384
  21--->La_Cure Le_Brassus 17
  22--->Baltimore Harrisburg 120
  23--->Yverdon Sainte_Croix 19
  24--->Aigle Bex 9
  25--->Oensingen Basel 48
  26--->Bex Saint_Maurice 4
  27--->Evian Saint_Gingolphe 17
  28--->Broc Jaun 19
  29--->Bulle Broc 4
  30--->Montbovon Les_Moulins 8.5
  31--->Montbovon Bulle 16.5
  32--->Chatel_Saint_Denis Vaulruz 13
  33--->Vaulruz Bulle 6
  34--->Mont_sur_Rolle Aubonne 7
  35--->Lausanne Tavernes 15
  36--->Syracuse Albany 217.6
  37--->Aubonne Biere 7
  38--->Payerne Grandcour 6
  39--->Divonne Gex 8
  40--->Olten Aarau 13
  41--->Neuchatel La_Chaux_de_Fonds 22
  42--->Gimel Biere 7
  43--->Basel Brugg 52
  44--->Biel Solothurn  24
  45--->Morgins Monthey 15
  46--->Biel Delemont 48
  47--->Saint_Gingolphe Chessel 9
  48--->Villars Salavaux 3
  49--->Avenches Salavaux 6
  50--->Thonon Evian 9
  51--->Vuarrens Orbe 12
  52--->Chessel Monthey 13.5
  53--->Bulle Riaz 3
  54--->Basel Brugg 52
  55--->Fribourg Le_Bry 15
  56--->Romont Fribourg 24
  57--->Echallens Vuarrens 5.5
  58--->Oron Romont 17
  59--->Prahins Yverdon 13
  60--->Harrisburg Buffalo 444.8
  61--->Washington Pittsburg 353.6
  62--->Pontarlier Morteau 31
```

63--->New_York Albany 246.4
64--->Morteau Le_Locle 15
65--->Les_Moulins Chateau_d'Oex 2
66--->Mollendruz Croy 15
67--->Bettens Orbe 14.5
68--->La_Sarraz Orbe 8.5
69--->Avenches Fribourg 15
70--->Salavaux Sugiez 9
71--->Cottens Cossonay 7
72--->Allaman Morges 9
73--->Ins Kerzers 8
74--->Bern Luzern 91
75--->Yverdon Neuchatel 38
76--->Grandcour Villars 7
77--->Aubonne Cottens 13
78--->Cottens Biere 11
79--->Morat Sugiez 6
80--->Luzern Olten 53
81--->Le_Sepey Les_Mosses 8
82--->Grandsivaz Fribourg 12
83--->Les_Mosses Les_Moulins 14
84--->Les_Mosses Chateau_d'Oex 15
85--->Lucens Payerne 16.5
86--->Payerne Romont 17.5
87--->Romont Vaulruz 12
88--->Bettens Echallens 11
89--->Carrouge Moudon 6.5
90--->Le_Sepey Leysin 7
91--->Massongex Bex 2
92--->Albany Boston 260.8
93--->Cheseaux Echallens 7
94--->Marchairuz Biere 11
95--->Col_du_Pillon Gstaad 17.5
96--->Gstaad Saanen 3.5
97--->Payerne Avenches 11
98--->La_Chaux_de_Fonds Le_Locle 9
99--->Rolle Allaman 5
100--->Aarau Brugg 19
101--->Moudon Lucens 5.5
102--->Le_Brassus Le_Pont 13
103--->Pittsburg Cleveland 206.4
104--->Rolle Mont_sur_Rolle 1
105--->Harrisburg Philadelphie 192
106--->Nyon Rolle 12
107--->Pittsburg Harrisburg 302.4
108--->Fontany Abondance 10
109--->Penthalaz Cossonay 2
110--->Penthalaz Bettens 6
111--->Nyon Saint_Georges 22
112--->Saint_Georges Gimel 5
113--->Payerne Grandsivaz 9
114--->Villars Avenches 4
115--->Chateau_d'Oex Saanen 12
116--->Montreux Villeneuve 5
117--->Ollon Villars_sur_Ollon 10
118--->Villars_sur_Ollon Col_de_la_Croix 8.5
119--->Le_Sepey Diablerets 8.5
120--->Diablerets Col_du_Pillon 5.5
121--->Cleveland Buffalo 309.2
122--->Saint_Georges Marchairuz 7
123--->Avenches Morat 8
124--->Brugg Baden 10
125--->Estavayer Grandcour 7
126--->Harrisburg Cleveland 516.8

127--->Pittsburg Buffalo 345.6
128--->Abondance Chatel 11.5
129--->Morges Lausanne 11
130--->Lausanne Penthalaz 14
131--->Neuchatel Le_Locle 29
132--->Harrisburg Scranton 188.8
133--->Washington Baltimore 59.2
134--->Fleurier Le_Locle 28
135--->Estavayer Payerne 10
136--->Saanen Zweisimmen 14
137--->Zweisimmen Boltigen 10
138--->Jaunpass Boltigen 10
139--->Jaun Jaunpass 7
140--->Vevey Montreux 7
141--->Lausanne Possens 29
142--->Morges Cottens 9
143--->Sainte_Croix Fleurier 13
144--->Pontarlier Fleurier 21
145--->Orbe Vallorbe 18
146--->Fribourg Le_Mouret 9
147--->Romont Le_Bry 16
148--->Le_Bry Riaz 9
149--->Riaz La_Roche 9
150--->Sugiez Ins 5
151--->Scranton Syracuse 217.6
152--->Vuarrens Yverdon 11
153--->Sainte_Croix Pontarlier 20
154--->Tavernes Vevey 16
155--->Nyon Divonne 10
156--->Baltimore Philadelphie 153.6
157--->Mont_sur_Rolle Gimel 8
158--->Mollendruz Le_Pont 2.5
159--->Vevey Chatel_Saint_Denis 12
160--->Lausanne Cheseaux 8.5
161--->Cheseaux Bettens 6
162--->Carrouge Tavernes 6.5
163--->Buffalo Syracuse 235.2
164--->Croy Orbe 6
165--->Zug Luzern 25
166--->Biere L'Isle 12
167--->Lausanne Vevey 18
168--->Rennaz Aigle 7
169--->L'Isle Mollendruz 9.5
170--->Oensingen Olten 15
171--->Scranton Buffalo 393.6
172--->New_York Scranton 220.8
173--->Scranton Albany 276.8
174--->Cossonay L'Isle 9
175--->Oron Chatel_Saint_Denis 10.5
176--->Col_de_la_Croix Diablerets 8.5
177--->Thierrens Prahins 4
178--->Monthey Massongex 3.5
179--->Douvaine Thonon 16
180--->Cossonay La_Sarraz 5.5
181--->Nyon La_Cure 22
182--->Geneve Gex 17
183--->Prahins Estavayer 17
184--->Bex Villars_sur_Ollon 15
185--->Aigle Ollon 5
186--->Croy Vallorbe 14
187--->Chessel Rennaz 4
188--->Villeneuve Rennaz 3
189--->Thonon Fontany 17.5
190--->Marchairuz Le_Brassus 7

```
191--->Morat Kerzers 9
192--->Solothurn Oensingen 17
193--->Fleurier Neuchatel 31
194--->Aigle Le_Sepey 10
195--->Echallens Possens 7.5
196--->Gex La_Cure 14
197--->La_Sarraz Croy 6.5
198--->Lausanne Carrouge 17.5
199--->Evian Fontany 18
200--->Orbe Yverdon 13
201--->Le_Mouret La_Roche 6
202--->Possens Moudon 10
203--->Neuchatel Ins 15
204--->Yverdon Estavayer 19
205--->Washington Harrisburg 171.2
206--->Aubonne Gimel 7
207--->Chatel Morgins 4.5
208--->Delemont Basel 41
209--->Nyon Mont_sur_Rolle 13
210--->Geneve Annemasse 7
211--->Echallens Prahins 16
212--->Possens Thierrens 7
213--->Moudon Thierrens 7
214--->Baden Zurich 34
215--->Zurich Zug 29
216--->Vallorbe Pontarlier 26
```

<<<FIN>>>

Par rapport à l'exemple traité au chapitre précédent, il a été rajouté un troisième sous-réseau connexe situé principalement en Suisse alémanique; ce sous-réseau est disjoint des deux autres.

Les résultats subséquents montrent que l'on a commencé par purger du réseau les noeuds situés dans un des sous-réseaux ne contenant pas la localité à partir de laquelle l'on forme les itinéraires. Cette purge a lieu dans un ordre arbitraire; les sous-réseaux purgés sont rassemblés dans une table séparée, ce qui permet d'établir les itinéraires depuis une de leurs localités. Pour chaque ensemble d'itinéraires, les localités concernées apparaissent dans l'ordre alphabétique, ce qui rend le dépouillement beaucoup plus aisé. On remarque que ce programme recherche, et liste dans l'ordre alphabétique, les localités situées dans un voisinage donné de la localité origine.

```
---Localites inaccessibles depuis Lausanne ---
Zurich  Zug  Washington  Solothurn  Scranton  Pittsburg  Philadelphie
Olten  Oensingen  New York  Luzern  Harrisburg  Delemont  Cleveland
Buffalo  Brugg  Boston  Biel  Bern  Basel  Baltimore  Albany  Aarau
Syracuse  Baden

***Itineraires depuis Lausanne***

   Abondance              81.5Km. via Chatel et route 128
   Aigle                  40.0Km. via Rennaz et route 168
   Allaman                20.0Km. via Morges et route  72
   Annemasse              67.0Km. via Geneve et route 210
   Aubonne                23.0Km. via Allaman et route  12
   Avenches               57.0Km. via Payerne et route  97
   Bettens                14.5Km. via Cheseaux et route 161
   Bex                    49.0Km. via Aigle et route  24
   Biere                  30.0Km. via Aubonne et route  37
   Boltigen               80.0Km. via Jaunpass et route 138
   Broc                   44.0Km. via Bulle et route  29
   Bulle                  40.0Km. via Vaulruz et route  33
   Carrouge               17.5Km. via route 198
   Chateau d'Oex          67.0Km. via Les Moulins et route  65
   Chatel                 70.0Km. via Morgins et route 207
   Chatel Saint Denis     29.5Km. via Oron et route 175
   Cheseaux                8.5Km. via route 160
   Chessel                37.0Km. via Rennaz et route 187
   Col de la Croix        63.5Km. via Villars sur Ollon et route 118
   Col du Pillon          64.0Km. via Diablerets et route 120
   Cossonay               16.0Km. via Penthalaz et route 109
   Cottens                20.0Km. via Morges et route 142
   Croy                   28.0Km. via La Sarraz et route 197
   Diablerets             58.5Km. via Le Sepey et route 119
   Divonne                47.0Km. via Nyon et route 155
   Douvaine               77.0Km. via Geneve et route  11
   Echallens              15.5Km. via Cheseaux et route  93
   Estavayer              48.5Km. via Prahins et route 183
   Evian                  63.0Km. via Saint Gingolphe et route  27
   Fleurier               64.0Km. via Sainte Croix et route 143
   Fontany                81.0Km. via Evian et route 199
   Fribourg               60.0Km. via Romont et route  56
   Geneve                 60.0Km. via Nyon et route   7
   Gex                    55.0Km. via Divonne et route  39
   Gimel                  30.0Km. via Aubonne et route 206
   Grandcour              52.0Km. via Payerne et route  38
   Grandsivaz             54.0Km. via Romont et route   1
   Gstaad                 81.5Km. via Col du Pillon et route  95
   Ins                    76.0Km. via Sugiez et route 150
   Jaun                   63.0Km. via Broc et route  28
   Jaunpass               70.0Km. via Jaun et route 139
   Kerzers                74.0Km. via Morat et route 191
   L'Isle                 25.0Km. via Cossonay et route 174
   La Chaux de Fonds      92.0Km. via Neuchatel et route  41
   La Cure                59.0Km. via Nyon et route 181
   La Roche               52.0Km. via Riaz et route 149
   La Sarraz              21.5Km. via Cossonay et route 180
   Lausanne             ***origine du parcours***
   Le Brassus             48.0Km. via Marchairuz et route 190
   Le Bry                 52.0Km. via Riaz et route 148
   Le Locle               92.0Km. via Fleurier et route 134
   Le Mouret              58.0Km. via La Roche et route 201
   Le Pont                37.0Km. via Mollendruz et route 158
   Le Sepey               50.0Km. via Aigle et route 194
   Les Mosses             58.0Km. via Le Sepey et route  81
```

Les Moulins	65.0Km. via Montbovon et route 30
Leysin	57.0Km. via Le Sepey et route 90
Lucens	29.5Km. via Moudon et route 101
Marchairuz	41.0Km. via Biere et route 94
Massongex	51.0Km. via Bex et route 91
Mollendruz	34.5Km. via L'Isle et route 169
Mont sur Rolle	26.0Km. via Rolle et route 104
Montbovon	56.5Km. via Bulle et route 31
Monthey	50.5Km. via Chessel et route 52
Montreux	25.0Km. via Vevey et route 140
Morat	65.0Km. via Avenches et route 123
Morges	11.0Km. via route 129
Morgins	65.5Km. via Monthey et route 45
Morteau	99.0Km. via Pontarlier et route 62
Moudon	24.0Km. via Carrouge et route 89
Neuchatel	70.0Km. via Yverdon et route 75
Nyon	37.0Km. via Rolle et route 106
Ollon	45.0Km. via Aigle et route 185
Orbe	29.0Km. via Bettens et route 67
Oron	19.0Km. via Tavernes et route 17
Payerne	46.0Km. via Lucens et route 85
Penthalaz	14.0Km. via route 130
Pontarlier	68.0Km. via Vallorbe et route 216
Possens	23.0Km. via Echallens et route 195
Prahins	31.5Km. via Echallens et route 211
Rennaz	33.0Km. via Villeneuve et route 188
Riaz	43.0Km. via Bulle et route 53
Rolle	25.0Km. via Allaman et route 99
Romont	36.0Km. via Oron et route 58
Saanen	79.0Km. via Chateau d'Oex et route 115
Saint Georges	35.0Km. via Gimel et route 112
Saint Gingolphe	46.0Km. via Chessel et route 47
Saint Maurice	53.0Km. via Bex et route 26
Sainte Croix	51.0Km. via Yverdon et route 23
Salavaux	62.0Km. via Villars et route 48
Sugiez	71.0Km. via Salavaux et route 70
Tavernes	15.0Km. via route 35
Thierrens	30.0Km. via Possens et route 212
Thonon	72.0Km. via Evian et route 50
Vallorbe	42.0Km. via Croy et route 186
Vaulruz	34.0Km. via Oron et route 16
Vevey	18.0Km. via route 167
Villars	59.0Km. via Grandcour et route 76
Villars sur Ollon	55.0Km. via Ollon et route 117
Villeneuve	30.0Km. via Montreux et route 116
Vuarrens	21.0Km. via Echallens et route 57
Yverdon	32.0Km. via Vuarrens et route 152
Zweisimmen	90.0Km. via Boltigen et route 137

+++Localites dans un rayon de 60.0Km. de Lausanne+++
Aigle Allaman Aubonne Avenches Bettens Bex Biere Broc
Bulle Carrouge Chatel Saint Denis Cheseaux Chessel Cossonay
Cottens Croy Diablerets Divonne Echallens Estavayer Fribourg
Geneve Gex Gimel Grandcour Grandsivaz L'Isle La Cure La Roche
La Sarraz Le Brassus Le Bry Le Mouret Le Pont Le Sepey Les Mosses
Leysin Lucens Marchairuz Massongex Mollendruz Mont sur Rolle
Montbovon Monthey Montreux Morges Moudon Nyon Ollon Orbe
Oron Payerne Penthalaz Possens Prahins Rennaz Riaz Rolle
Romont Saint Georges Saint Gingolphe Saint Maurice Sainte Croix
Tavernes Thierrens Vallorbe Vaulruz Vevey Villars Villars sur Ollon
Villeneuve Vuarrens Yverdon
<<<FIN>>>

+++Tout le reseau considere accessible depuis Geneve +++

Itineraires depuis Geneve

Abondance	60.5Km.	via Fontany et route 108
Aigle	79.0Km.	via Rennaz et route 168
Allaman	40.0Km.	via Rolle et route 99
Annemasse	7.0Km.	via route 210
Aubonne	43.0Km.	via Mont sur Rolle et route 34
Avenches	117.0Km.	via Payerne et route 97
Bettens	69.0Km.	via Penthalaz et route 110
Bex	87.0Km.	via Massongex et route 91
Biere	50.0Km.	via Aubonne et route 37
Boltigen	140.0Km.	via Jaunpass et route 138
Broc	104.0Km.	via Bulle et route 29
Bulle	100.0Km.	via Vaulruz et route 33
Carrouge	77.5Km.	via Lausanne et route 198
Chateau d'Oex	112.0Km.	via Les Mosses et route 84
Chatel	72.0Km.	via Abondance et route 128
Chatel Saint Denis	89.5Km.	via Oron et route 175
Cheseaux	68.5Km.	via Lausanne et route 160
Chessel	68.0Km.	via Saint Gingolphe et route 47
Col de la Croix	102.5Km.	via Villars sur Ollon et route 118
Col du Pillon	103.0Km.	via Diablerets et route 120
Cossonay	61.0Km.	via Morges et route 2
Cottens	56.0Km.	via Aubonne et route 77
Croy	73.0Km.	via La Sarraz et route 197
Diablerets	97.5Km.	via Le Sepey et route 119
Divonne	25.0Km.	via Gex et route 39
Douvaine	17.0Km.	via route 11
Echallens	75.5Km.	via Cheseaux et route 93
Estavayer	107.0Km.	via Yverdon et route 204
Evian	42.0Km.	via Thonon et route 50
Fleurier	119.0Km.	via Pontarlier et route 144
Fontany	50.5Km.	via Thonon et route 189
Fribourg	120.0Km.	via Romont et route 56
Geneve	***origine du parcours***	
Gex	17.0Km.	via route 182
Gimel	44.0Km.	via Mont sur Rolle et route 157
Grandcour	112.0Km.	via Payerne et route 38
Grandsivaz	114.0Km.	via Romont et route 1
Gstaad	120.5Km.	via Col du Pillon et route 95
Ins	136.0Km.	via Sugiez et route 150
Jaun	123.0Km.	via Broc et route 28
Jaunpass	130.0Km.	via Jaun et route 139
Kerzers	134.0Km.	via Morat et route 191
L'Isle	62.0Km.	via Biere et route 166
La Chaux de Fonds	148.0Km.	via Neuchatel et route 41
La Cure	31.0Km.	via Gex et route 196
La Roche	112.0Km.	via Riaz et route 149
La Sarraz	66.5Km.	via Cossonay et route 180
Lausanne	60.0Km.	via Morges et route 129
Le Brassus	48.0Km.	via La Cure et route 21
Le Bry	112.0Km.	via Riaz et route 148
Le Locle	144.0Km.	via Morteau et route 64
Le Mouret	118.0Km.	via La Roche et route 201
Le Pont	61.0Km.	via Le Brassus et route 102
Le Sepey	89.0Km.	via Aigle et route 194
Les Mosses	97.0Km.	via Le Sepey et route 81
Les Moulins	111.0Km.	via Les Mosses et route 83
Leysin	96.0Km.	via Le Sepey et route 90
Lucens	89.5Km.	via Moudon et route 101
Marchairuz	52.0Km.	via Saint Georges et route 122

Massongex	85.0Km.	via Monthey et route 178
Mollendruz	63.5Km.	via Le Pont et route 158
Mont sur Rolle	36.0Km.	via Nyon et route 209
Montbovon	116.5Km.	via Bulle et route 31
Monthey	81.5Km.	via Chessel et route 52
Montreux	80.0Km.	via Villeneuve et route 116
Morat	125.0Km.	via Avenches et route 123
Morges	49.0Km.	via Allaman et route 72
Morgins	76.5Km.	via Chatel et route 207
Morteau	129.0Km.	via Pontarlier et route 62
Moudon	84.0Km.	via Carrouge et route 89
Neuchatel	126.0Km.	via Yverdon et route 75
Nyon	23.0Km.	via route 7
Ollon	84.0Km.	via Aigle et route 185
Orbe	75.0Km.	via La Sarraz et route 68
Oron	79.0Km.	via Tavernes et route 17
Payerne	106.0Km.	via Lucens et route 85
Penthalaz	63.0Km.	via Cossonay et route 109
Pontarlier	98.0Km.	via Vallorbe et route 216
Possens	83.0Km.	via Echallens et route 195
Prahins	91.5Km.	via Echallens et route 211
Rennaz	72.0Km.	via Chessel et route 187
Riaz	103.0Km.	via Bulle et route 53
Rolle	35.0Km.	via Nyon et route 106
Romont	96.0Km.	via Oron et route 58
Saanen	124.0Km.	via Gstaad et route 96
Saint Georges	45.0Km.	via Nyon et route 111
Saint Gingolphe	59.0Km.	via Evian et route 27
Saint Maurice	88.0Km.	via Massongex et route 5
Sainte Croix	107.0Km.	via Yverdon et route 23
Salavaux	122.0Km.	via Villars et route 48
Sugiez	131.0Km.	via Salavaux et route 70
Tavernes	75.0Km.	via Lausanne et route 35
Thierrens	90.0Km.	via Possens et route 212
Thonon	33.0Km.	via Douvaine et route 179
Vallorbe	72.0Km.	via Le Pont et route 6
Vaulruz	94.0Km.	via Oron et route 16
Vevey	78.0Km.	via Lausanne et route 167
Villars	119.0Km.	via Grandcour et route 76
Villars sur Ollon	94.0Km.	via Ollon et route 117
Villeneuve	75.0Km.	via Rennaz et route 188
Vuarrens	81.0Km.	via Echallens et route 57
Yverdon	88.0Km.	via Orbe et route 200
Zweisimmen	138.0Km.	via Saanen et route 136

+++Localites dans un rayon de 75.0Km. de Geneve+++
Abondance Allaman Annemasse Aubonne Bettens Biere Chatel
Cheseaux Chessel Cossonay Cottens Croy Divonne Douvaine
Evian Fontany Gex Gimel L'Isle La Cure La Sarraz Lausanne
Le Brassus Le Pont Marchairuz Mollendruz Mont sur Rolle Morges
Nyon Orbe Penthalaz Rennaz Rolle Saint Georges Saint Gingolphe
Tavernes Thonon Vallorbe Villeneuve
<<<FIN>>>

---Localites inaccessibles depuis Luzern ---
Harrisburg Baltimore Albany Buffalo Cleveland Boston Scranton
New York Pittsburg Philadelphie Washington Syracuse

Itineraires depuis Luzern

```
   Aarau              66.0Km. via Olten et route  40
   Baden              88.0Km. via Zurich et route 214
   Basel             116.0Km. via Oensingen et route  25
   Bern               91.0Km. via route 74
   Biel              109.0Km. via Solothurn et route  44
   Brugg              85.0Km. via Aarau et route 100
   Delemont          157.0Km. via Biel et route  46
   Luzern           ***origine du parcours***
   Oensingen          68.0Km. via Olten et route 170
   Olten              53.0Km. via route  80
   Solothurn          85.0Km. via Oensingen et route 192
   Zug                25.0Km. via route 165
   Zurich             54.0Km. via Zug et route 215

+++Localites dans un rayon de  65.0Km. de Luzern+++
Olten  Zug  Zurich
<<<FIN>>>

---Localites inaccessibles depuis Bern ---
Harrisburg  Baltimore  Albany  Buffalo  Cleveland  Boston  Scranton
New York  Pittsburg  Philadelphie  Washington  Syracuse

***Itineraires depuis Bern***

   Aarau              81.0Km. via Olten et route  40
   Baden             110.0Km. via Brugg et route 124
   Basel             101.0Km. via Oensingen et route  25
   Bern             ***origine du parcours***
   Biel               33.0Km. via route  9
   Brugg             100.0Km. via Aarau et route 100
   Delemont           81.0Km. via Biel et route  46
   Luzern             91.0Km. via route  74
   Oensingen          53.0Km. via Solothurn et route 192
   Olten              68.0Km. via Oensingen et route 170
   Solothurn          36.0Km. via route  8
   Zug               116.0Km. via Luzern et route 165
   Zurich            144.0Km. via Baden et route 214

---Toutes les localites sont a plus de  30.0Km. de Bern---
<<<FIN>>>

---Localites inaccessibles depuis New York ---
Biel  Aarau  Baden  Bern  Basel  Brugg  Delemont  Olten  Luzern
Oensingen  Solothurn  Zurich  Zug

***Itineraires depuis New York***

   Albany            246.4Km. via route  63
   Baltimore         313.6Km. via Philadelphie et route 156
   Boston            384.0Km. via route  20
   Buffalo           614.4Km. via Scranton et route 171
   Cleveland         860.8Km. via Pittsburg et route 103
   Harrisburg        352.0Km. via Philadelphie et route 105
   New York         ***origine du parcours***
   Philadelphie      160.0Km. via route  14
   Pittsburg         654.4Km. via Harrisburg et route 107
   Scranton          220.8Km. via route 172
   Syracuse          438.4Km. via Scranton et route 151
   Washington        372.8Km. via Baltimore et route 133

---Toutes les localites sont a plus de 120.0Km. de New York---
<<<FIN>>>

<<<<<FIN DES APPLICATIONS>>>>>
```

La structure générale du programme *villes* modifié est donnée à la figure 52.

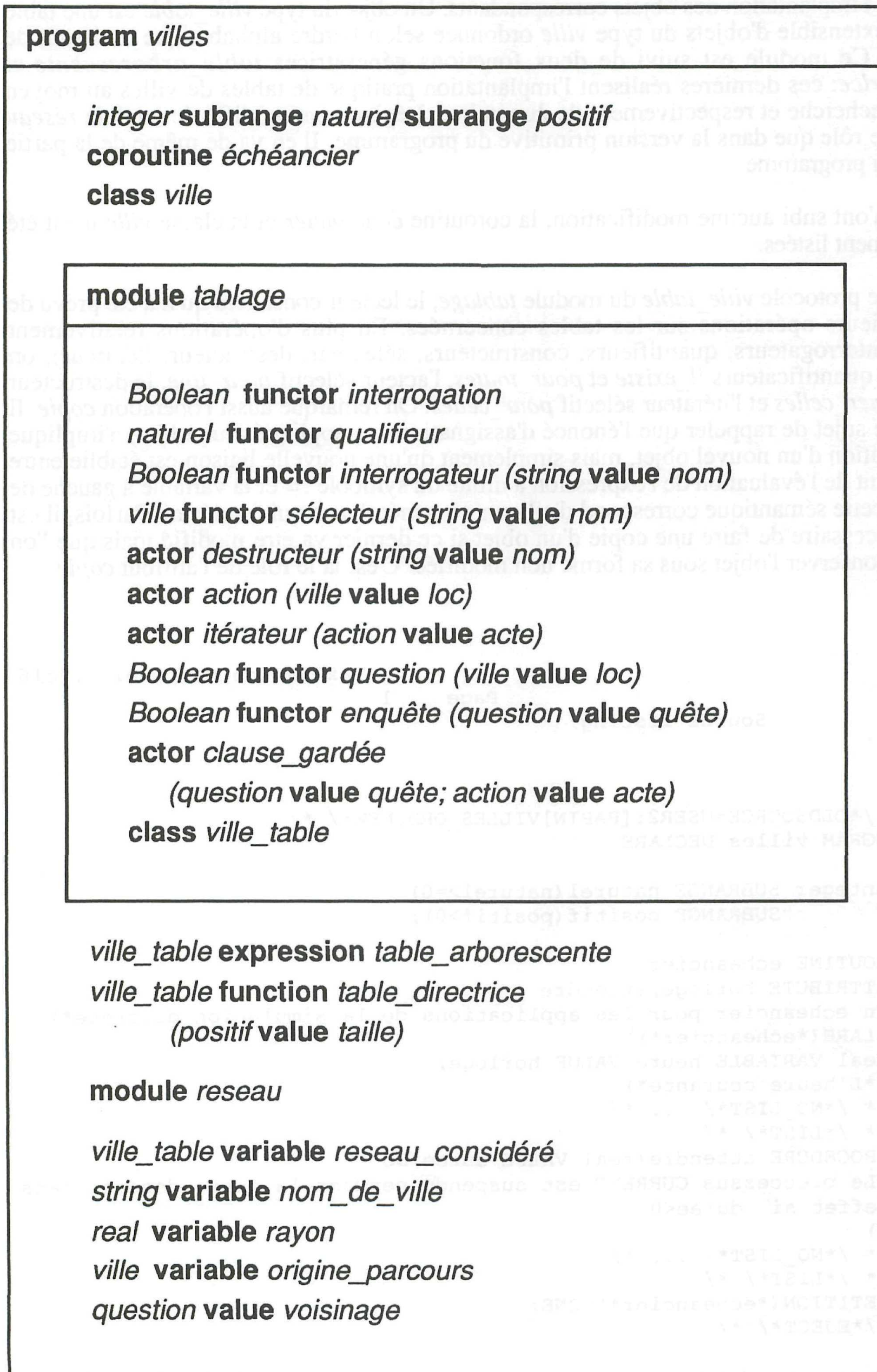


Fig. 52

La coroutine *échéancier* et la classe *ville* jouent le même rôle que dans le programme primitif. Le module *tablage* introduit la classe protocole *ville_table,* ainsi que les types procéduraux nécessaires à l'implantation des objets correspondants. Un objet du type *ville_table* est une table associative extensible d'objets du type *ville* ordonnée selon l'ordre alphabétique croissant de leurs noms. Ce module est suivi de deux fonctions génératrices *table_arborescente* et *table_directrice*: ces dernières réalisent l'implantation pratique de tables de villes au moyen d'arbres de recherche et respectivement de directoires. Légèrement modifié, le module *réseau* joue le même rôle que dans la version primitive du programme. Il en va de même de la partie exécutable du programme

Vu qu'elles n'ont subi aucune modification, la coroutine *échéancier* et la classe *ville* n'ont été que partiellement listées.

Dans la classe protocole *ville_table* du module *tablage,* le lecteur constatera qu'il a été prévu de réaliser plusieurs opérations sur les tables concernées. En plus d'opérations relativement classiques (interrogateurs, quantifieurs, constructeurs, sélecteur, destructeur, itérateur), on remarque les quantificateurs *il_existe* et *pour_toutes,* l'acteur sélectif *pour_une,* le destructeur sélectif *éliminer_celles* et l'itérateur sélectif *pour_celles.* On remarque aussi l'opération *copie.* Il convient à ce sujet de rappeler que l'énoncé d'assignation *:=,* appliqué à un objet, n'implique jamais la création d'un nouvel objet, mais simplement qu'une nouvelle liaison est établie entre l'objet résultant de l'évaluation de l'expression à droite du symbole := et la variable à gauche de ce symbole; cette sémantique correspond d'ailleurs en général à ce qui est désiré. Parfois, il est cependant nécessaire de faire une copie d'un objet si ce dernier va être modifié mais que l'on désire aussi conserver l'objet sous sa forme non modifiée. C'est là le rôle de l'attribut *copie.*

```
villes                                          Vax Newton Compiler 0.2c16
                                          Page    1
                   Source listing

/* /*OLDSOURCE=USER2:[RAPIN]VILLES_ORD.NEW*/ */
PROGRAM villes DECLARE

  integer SUBRANGE naturel(naturel>=0)
          SUBRANGE positif(positif>0);

COROUTINE echeancier
  ATTRIBUTE horloge,attendre
  (*Un echeancier pour les applications de la simulation discrete*)
DECLARE(*echeancier*)
  real VARIABLE heure VALUE horloge;
  (*L'heure courante*)
  /* /*NO_LIST*/ ... */
  /* /*LIST*/ */
  PROCEDURE attendre(real VALUE duree)DO
 (*Le preocessus CURRENT est suspendu pendant la duree donnee; sans
   effet si  duree<0
  *)
  /* /*NO_LIST*/ ... */
  /* /*LIST*/ */
REPETITION(*echeancier*)DONE;
/* /*EJECT*/ */
```

```
villes                                          Vax Newton Compiler 0.2c16
                                           Page    2
                 Source listing

     CLASS ville
       VALUE moi
       ATTRIBUTE nom,relier,atteinte,etape,itineraire,distance,
                  init,origine
       (string VALUE nom)
     DECLARE(*ville*)
       /* /*NO_LIST*/ ... */
       /* /*LIST*/ */
       ville FUNCTION relier
         (positif VALUE numero_route; real VALUE distance;
          ville VALUE destination)
       (*Relie la ville concernee a la ville  destination au moyen
         du troncon de route  numero_route  de longueur  distance
       *)
       /* /*NO_LIST*/ ... */
       /* /*LIST*/ */
       Boolean VARIABLE att VALUE atteinte:=FALSE;
       (*Vrai ssi la ville concernee a ete atteinte par un courrier*)

       ville VARIABLE etp VALUE etape;
       (*La ville precedante du parcours menant a la ville concernee.

         Condition d'emploi:  atteinte
       *)

       naturel VARIABLE itin VALUE itineraire;
       (*Le troncon de route par lequel la ville concernee a ete
         atteinte.

         Condition d'emploi: atteinte
       *)

       real VARIABLE dist VALUE distance;
       (*La distance depuis la ville origine du parcours.

         Condition d'emploi:  atteinte
       *)
       /* /*NO_LIST*/ ... */
       /* /*LIST*/ */
       ville EXPRESSION init=
       (*reinitialise la ville en vue du calcul de sa distance depuis
         une ville arbitraire du reseau; le resultat est la ville
         concernee  moi .
       *)
       /* /*NO_LIST*/ ... */
       /* /*LIST*/ */
       ville EXPRESSION origine=
       (*recherche le plus court chemin menant de la ville concernee
         a chacune des villes du reseau; le resultat est la ville
         concernee  moi .

         Condition d'emploi:  init  a ete effectuee pour toutes les
```

```
villes                                              Vax Newton Compiler 0.2c16
                                          Page    3
                     Source listing

                villes du reseau au prealable.
             *)
             /* /*NO_LIST*/ ... */
             /* /*LIST*/ */
          DO(*ville*)DONE;
       /* /*EJECT*/ */
```

En regardant le texte du module *tablage* et de la classe *ville_table*, on remarque que *copie* est introduite sous la forme d'une expression formelle et non d'un objet procédural. Un tel objet procédural aurait dû appartenir à un type de la forme :

ville_table **functor** *générateur*

Ceci aurait manifestement impliqué une récursion mutuelle entre les types *ville_table* et *générateur*, ce qui ne peut être commodément exprimé dans l'état actuel du langage Newton : il n'est possible de déclarer le type *générateur* ni avant, ni après la classe *ville_table*. On verra que cette récursion mutuelle implique également certains problèmes lors de la conception des fonctions génératrices.

La fonction génératrice *table_arborescente* implante le type abstrait *ville_table* au moyen d'un arbre de recherche. Cette structure de données, très versatile, est utilisable pour l'implantation de tables associatives indicées par les valeurs d'un type de base ordonné. On va en rappeler ici le principe et décrire les algorithmes nécessaires à la réalisation des principales opérations. Supposant, par exemple, que le type des indices soit le type *string*, un arbre de recherche est un arbre binaire de la forme :

object *arbre*
    (string **value** *membre;*
    arbre **variable** *gauche, droite)*
    variable *racine* := **nil**

On a supposé ici un arbre *racine* initialement vide; bien entendu, à chaque noeud d'un tel arbre, il sera en général défini d'autres informations associées à la chaîne indiçante *membre*.

Soit *a* un arbre de recherche. Soit *a* est vide; dans le cas contraire, *a* est un triple *(a.membre, a.gauche, a.droite)*. Dans ce dernier cas, tous les membres du sous-arbre *a.gauche* sont inférieurs au membre privilégié *a.membre*; tous les membres du sous-arbre *a.droite* lui sont supérieurs (Fig. 53). L'arbre de la figure 54 n'est, par contre, pas un arbre de recherche : la propriété énoncées est bien satisfaite pour l'arbre principal *racine* mais non pour le noeud de membre *Paul* qui contient l'élément *René* dans son sous-arbre gauche.

```
MODULE tablage
  ATTRIBUTE
    ville_table,
    interrogation,quantifieur,interrogateur,selecteur,destructeur,
    action,iterateur,question,enquete,clause_gardee
 (*ce module definit la classe protocole  ville_table  representant
   l'interface de tables associatives  de villes ordonnees selon
   l'ordre alphabetique de leurs noms ainsi que les types procedu-
   raux necessaires a leur implantation
 *)
DECLARE(*tablage*)
  Boolean FUNCTOR interrogation;
  naturel FUNCTOR quantifieur;
  Boolean FUNCTOR interrogateur(string VALUE nom);
  ville FUNCTOR selecteur(string VALUE nom);
  ACTOR destructeur(string VALUE nom);
  ACTOR action(ville VALUE loc);
  ACTOR iterateur(action VALUE acte);
  Boolean FUNCTOR question(ville VALUE loc);
  Boolean FUNCTOR enquete(question VALUE quete);
  ACTOR clause_gardee
    (question VALUE quete; action VALUE acte);

  CLASS ville_table
    VALUE moi
    INDEX composante
    ATTRIBUTE
      copie,vide,quantite,contient,inserer,eliminer,parcourir,
      il_existe,pour_toutes,pour_une,pour_celles,eliminer_celles
    (ville_table EXPRESSION copie;
     interrogation VALUE vd;
     quantifieur VALUE quant;
     interrogateur VALUE cont;
     action VALUE ins;
     selecteur VALUE sel;
     destructeur VALUE des;
     iterateur VALUE parc;
     enquete VALUE une,toutes;
     clause_gardee VALUE pr_une,pr_celles,el_celles)
   (*Un objet du type  ville_table  est une table, initialement vide
     de villes. L'implanteur fournira, sous la forme d'objets proce-
     duraux, les operations suivantes:

       copie                  une copie de la table concernee
       vd EVAL                vrai ssi la table est vide
       quant EVAL             le nombre d'elements de la table
       cont[nom]              vrai ssi la table contient une ville
                              baptisee  nom
       ins[loc]               insere dans la table la ville  loc ;
                              sans effet si cette ville s'y trouve
                              deja
       sel[nom]               recherche dans la table une ville
```

```
villes                                  Vax Newton Compiler 0.2c16
                                  Page    5
               Source listing

  575                          baptisee  nom  et retourne cette der-
  575                          niere; si la table ne possede aucune
  575                          telle composante, cree un nouvel objet
  575                          du type  ville  et l'y insere dans la
  575                          table
  575       des[nom]           elimine de la table la ville baptisee
  575                          nom ; sans-effet si la table ne contient
  575                          aucune ville du nom approprie
  575       parc[acte]         effectue l'operation  acte[loc]  pour
  575                          chaque ville  loc  de la table; les
  575                          villes seront traitees dans l'ordre
  575                          alphabetique croissant de leur nom
  575       une[cond]          vrai ssi la table possede une ville
  575                          loc  (au moins) pour laquelle  cond[loc]
  575                          est vraie
  575       touts[cond]        vrai ssi l'expression  cond[loc]  est
  575                          vraie pour chaque ville  loc  de la
  575                          table
  575       pr_une[cond,acte]  effectue l'enonce  acte[loc]  pour l'une
  575                          des localites satisfaisant a la condi-
  575                          tion  cond[loc]
  575       pr_celles[cond,acte] effectue, dans l'ordre alphabetique
  575                         acte[loc]  pour chaque ville  loc  satis-
  575                          faisant a la condition  cond[loc]
  575       el_celles[cond,acte] elimine de la table toutes les villes
  575                          loc  satisfaisant a la condition
  575                          cond[loc] ; effectue pour chacune
  575                          d'entres elles l'enonce  acte[loc]
  575
  575    *)
  575    DECLARE(*ville_table*)
  576      Boolean EXPRESSION vide=(*vrai ssi la table est vide*)
  580        vd EVAL;
  583
  583      naturel EXPRESSION quantite=
  587      (*le nombre de villes dans la table*)
  587        quant EVAL;
  590
  590      Boolean FUNCTION contient
  593        (string VALUE nom)
  598      (*vrai ssi la table possede une ville baptisee  nom *)
  598      DO TAKE cont[nom] DONE;
  606
  606      ville_table FUNCTION inserer
  609        (ville VALUE localite)
  614      (*insere, dans la table concernee, la ville  localite ;
  614        sans effet si cette ville s'y trouve deja. Le resultat
  614        est la table concernee
  614      *)
  614      DO ins[localite] TAKE moi DONE;
  623
  623      ville FUNCTION composante
```

```
       (string VALUE nom)
      (*le resultat est l'element baptise  nom  de la table; si
        la table ne contient aucune telle composante, cree une
        nouvelle ville du nom donne et l'y insere
      *)
      DO TAKE sel[nom] DONE;

      ville_table FUNCTION eliminer
        (string VALUE nom)
      (*elimine de la table la localite baptisee  nom ; sans effet
        si  ~contient(nom) . Le resultat est la table concernee
        moi
      *)
      DO des[nom] TAKE moi DONE;

      ville_table FUNCTION parcourir
        (action VALUE acte)
      (*effectue, dans l'ordre alphabetique croissant de leur nom,
        l'enonce  acte[loc]  pour chaque ville  loc  de la table.
        Le resultat est la table concernee  moi .
      *)
      DO parc[acte] TAKE moi DONE;

      Boolean FUNCTION il_existe
        (question VALUE quete)
      (*vrai ssi la table possede une ville  loc  pour laquelle
        quete[loc]  est vraie
      *)
      DO TAKE une[quete] DONE;

      Boolean FUNCTION pour_toutes
        (question VALUE quete)
      (*vrai ssi l'expression  quete[loc]  est vraie pour chacune
        des villes  loc  de la table
      *)
      DO TAKE toutes[quete] DONE;

      ville_table FUNCTION pour_une
        (question VALUE quete; action VALUE acte)
      (*effectue l'enonce  acte[loc]  pour l'une des villes  loc
        satisfaisant a la condition  quete[loc] ; sans effet si
        la table ne possede aucune telle localite. Le resultat
        est la table concernee  moi
      *)
      DO pr_une[quete,acte] TAKE moi DONE;

      ville_table FUNCTION pour_celles
        (question VALUE quete; action VALUE acte)
      (*effectue, dans l'ordre alphabetique croissant de leurs
        noms, l'enonce  acte[loc]  pour chaque ville  loc  de
        la table satisfaisant a la condition  quete[loc] . Le
        resultat est la table concernee  moi
```

```
villes                                        Vax Newton Compiler 0.2c16
                                    Page    7
              Source listing

 740        *)
 740        DO pr_celles[quete,acte] TAKE moi DONE;
 751
 751        ville_table FUNCTION eliminer_celles
 754          (question VALUE quete; action VALUE acte)
 763        (*elimine de la table chaque ville  loc  satisfaisant a
 763          la condition  quete[loc] ; effectue l'enonce  acte[loc]
 763          pour chaque localite eliminee. L'ordre des eliminations
 763          n'est pas defini a priori. Le resultat est la table
 763          concernee  moi
 763        *)
 763        DO el_celles[quete,acte] TAKE moi DONE
 773      DO(*ville_table*)DONE
 775    DO(*tablage*)DONE;
 778    /* /*EJECT*/ */
```

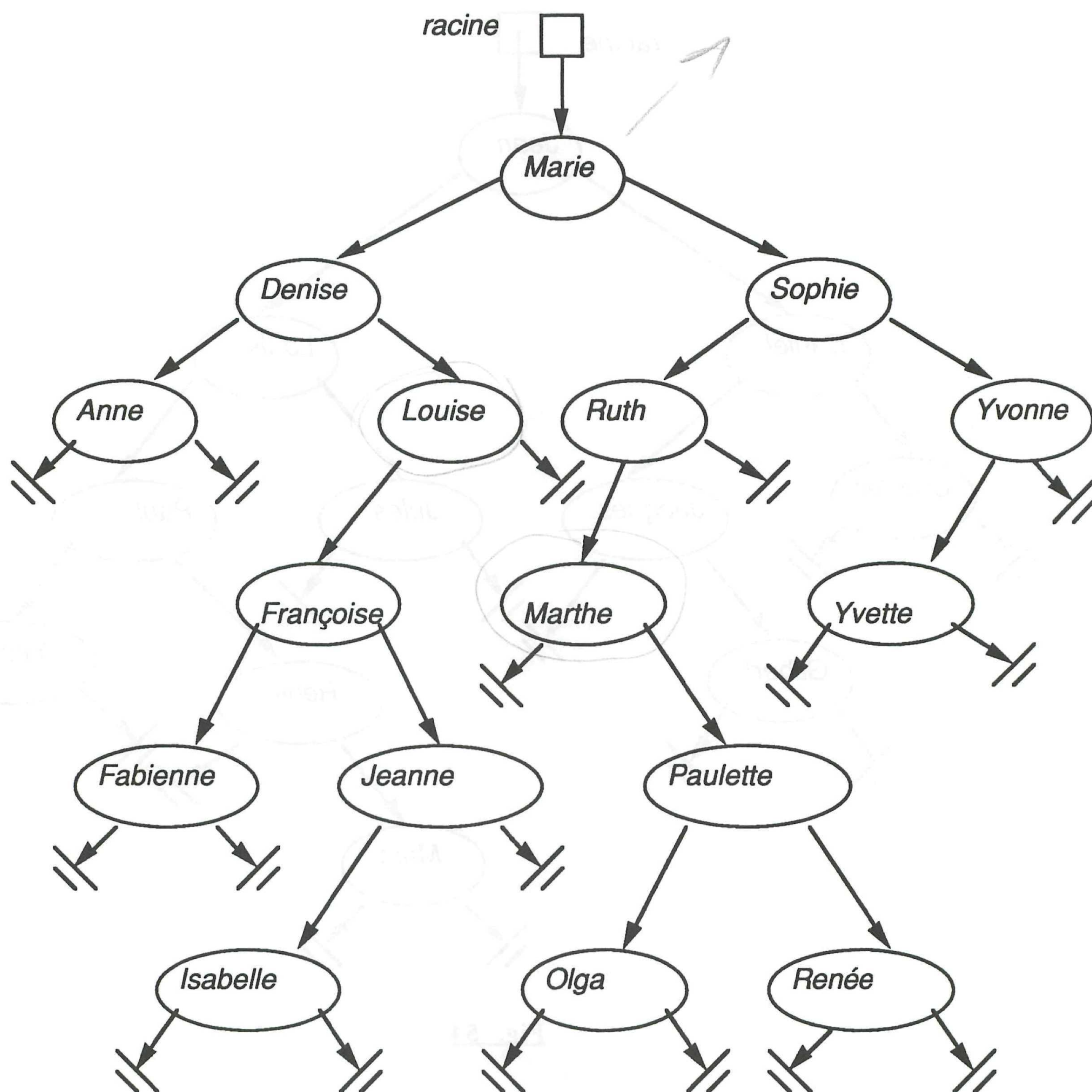

racine
Marie
Denise
Sophie
Anne
Louise
Ruth
Yvonne
Françoise
Marthe
Yvette
Fabienne
Jeanne
Paulette
Isabelle
Olga
Renée

Fig. 53

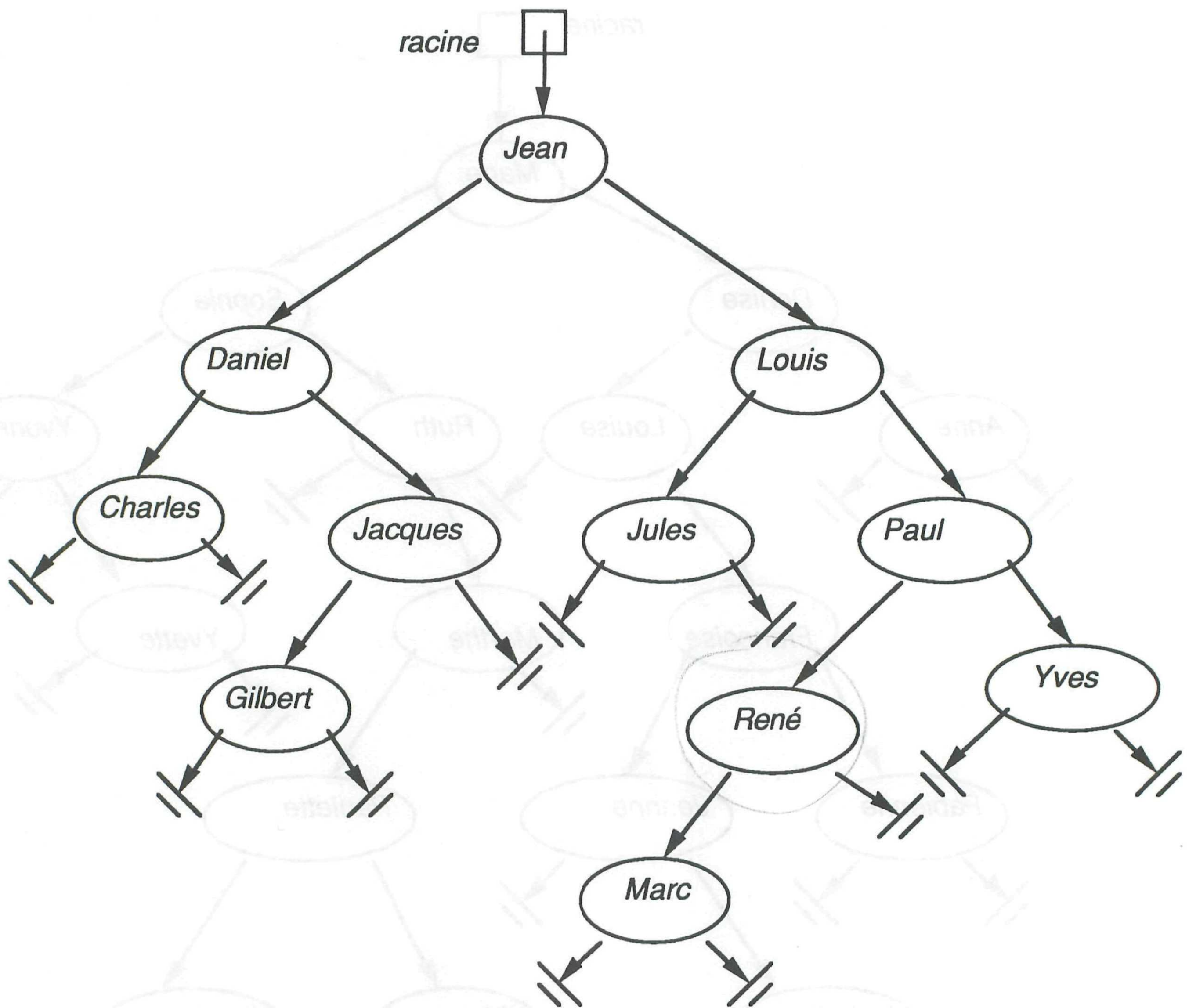


Fig. 54

Un arbre de recherche est une structure de données récursive; la plupart des algorithmes nécessaires à la réalisation des principales opérations sont de nature récursive. Dans certains cas (sélecteur, constructeur, destructeur), la récursion peut être éliminée par des moyens simples; dans d'autres cas, notamment pour l'itérateur, l'élimination de la récursion impliquerait l'usage d'une pile explicite : ces algorithmes-là seront laissés sous leur forme itérative.

Pour rechercher si un élément donné *nom* figure dans un arbre de recherche *a*, il suffit de considérer les cas suivants :

1. *a* = **nil** ; il est évident que *nom* ne fait pas partie de *a*

2. *nom* = *a.membre* ; il est évident que *nom* fait partie de *a*

3. *nom* < *a.membre* ; il suffit de rechercher *nom*, récursivement, dans le sous-arbre *a.gauche*

4. *nom* > *a.membre* ; il suffit de rechercher *nom*, récursivement, dans le sous-arbre *a.droite*

Ceci débouche sur l'algorithme suivant :

```
Boolean function contient
   (string value nom)
(* vrai si l'arbre issu de la variable racine contient le membre nom
*)
declare
   Boolean function cont
      (arbre value rac)
   do (*cout*) take
      connect rac then take
         if nom < membre then cont (gauche) else
         if nom > membre then cont (droite)
         default (*nom = membre*) true done
      default false done
   done (*cont*)
do take cont (racine) done
```

Cet algorithme satisfait clairement au théorème de dérécursification énoncé au chapitre 2 : les actions récursives interviennent à la fin du schéma d'exécution. On a la formulation itérative suivante :

```
Boolean function contient
   (string value nom)
(*vrai si l'arbre issu de la variable racine contient le membre nom *)
declare
   arbre variable rac := racine
do (*contient*)
   cycle cont do
   connect rac then
      if nom < membre then
         rac := gauche
      repeat cont done ;
      if nom > membre then
         rac := droite
      repeat cont done
   done (* connect rac*)
   done (* cycle cont*)
take rac ~= nil done
```

Dans cette formulation itérative, *rac* joue le rôle d'un curseur qui descend dans l'arbre jusqu'à ce que l'on trouve un noeud contenant l'élément de valeur *nom* ou que l'on tombe sur un noeud vide. Plutôt que d'implanter l'algorithme de recherche directement dans la fonction *contient*, il est préférable de l'isoler dans la fonction d'accès suivante :

```
arbre access cherche
   (string value nom)
(* On considère l'arbre issu de la variable racine; recherche si cet arbre contient un élément
   de valeur nom. Le cas échéant, résulte en un repère à la variable contenant le lien au noeud
   correspondant; dans le cas contraire, résulte en un repère à la variable dans laquelle un tel
   élément pourrait être inséré.
*)

declare arbre reference rac -> racine do
   cycle descend do
   connect rac then
      if nom < membre then
         rac -> gauche
      repeat descend done;
      if nom > membre then
         rac -> droite
```

```
        repeat descend done
    done (* connect rac *)
    done (* cycle descend *)
do take rac done
```

Le curseur est, cette fois-ci, une référence. L'avantage de passer par cette fonction d'accès est qu'elle facilite l'implantation de plusieurs opérations. La fonction *contient* prend évidemment la forme suivante :

```
Boolean function contient
    (string value nom)
do take cherche (nom)  ~= nil done
```

Il est également facile de réaliser un constructeur :

```
procedure placer
    (string value nom)
(* Insère, dans l'arbre issu de la variable racine, un nouvel élément de valeur nom. Sans
    effet si l'arbre contient déjà un tel membre
*)
declare
    arbre reference rac -> cherche (nom)
do
    if rac = nil then
        rac := arbre (nom, nil, nil)
    done
done (* placer *)
```

Appliqué à l'arbre de la figure 53 avec *nom = "Henriette"*, cet algorithme insérerait le nouveau noeud à gauche de celui contenant le membre *Isabelle*.

On peut montrer qu'en moyenne, cet algorithme permet de chercher ou d'insérer un élément dans un arbre de *n* membres en un temps proportionnel au logarithme de *n* . Le temps total de construction d'un tel arbre est alors de l'ordre de *n * ln (n)*. Il existe cependant des cas pathologiques où ce temps est nettement plus grand; c'est notamment le cas si les éléments y sont insérés par ordre croissant (ou décroissant). Dans ce cas, l'arbre dégénère en une liste triée construite par le bout : le temps nécessaire à l'insertion de l'élément de rang *k* est proportionnel à *k* et le temps total à la construction de l'arbre de *n* membres est de l'ordre de *n**2*. On notera, à ce sujet, qu'il y a toujours avantage à maintenir un arbre de recherche sous une forme relativement équilibrée si l'on veut éviter que les chemins d'accès à certains éléments soient exagérément longs.

L'élimination d'un élément d'un arbre de recherche est plus délicate; au moyen de la fonction d'accès *cherche*, on commence par se ramener au cas où l'élément à éliminer est à la racine de l'arbre :

```
procedure ôter (string value nom) declare
(*Elimine de l'arbre issu de la variable racine l'élément de valeur nom.
    Sans effet si l'arbre ne possède pas de tel élément.
*)
    arbre reference rac -> cherche (nom)
do (*ôter*)
    if rac ~= nil then
        déraciner (rac)
    done
done (*ôter*)
```

Pour réaliser la procédure *déraciner*, on commence par éliminer les deux cas triviaux où l'un des sous-arbres de l'arbre considéré est vide. Il reste donc le cas où ces deux sous-arbres sont non

vides. Une première possibilité, facile à programmer, consisterait à placer le sous-arbre droite à l'extrême droite du sous-arbre gauche ou vice-versa. Dans le cas de l'arbre représenté à la ligne 53, ceci reviendrait à placer le sous-arbre de racine *Sophie* à droite de l'élément *Louise* ou respectivement le sous-arbre de racine *Denise* à gauche de l'élément *Marthe*. Bien que fonctionnellement correcte, cette manière de faire est criticable : elle tend à déséquilibrer l'arbre et à rallonger certains chemins d'accès.

Partant de l'idée que l'arbre initial n'est pas trop déséquilibré, on a avantage à prendre pour nouvelle racine un élément de valeur pas trop différente de celle du membre que l'on a éliminé. Les deux éléments les plus voisins de ce dernier sont situés tout à gauche du sous-arbre droite (*Marthe* dans le cas de l'arbre de la figure 53) et tout à droite du sous-arbre gauche (*Louise* dans le cas de la figure 53). On va donc promouvoir comme racine l'un de ces deux éléments et rétablir les liaisons rompues de manière à conserver la structure d'arbre de recherche. Ainsi, pour promouvoir l'élément sité à gauche du sous-arbre droite, il faut établir les liaisons suivantes :

1. Réaccrocher le sous-arbre droite de l'élément que l'on promeut à l'emplacement d'où était issu cet élément.

2. Réaccrocher comme sous-arbres gauche et droite de l'élément que l'on promeut ceux du noeud qui a été éliminé.

3. Ne pas oublier de réaccrocher l'élément que l'on promeut à la racine de l'arbre.

La procédure *déraciner* peut être programmée comme suit :

```
procedure déraciner (arbre reference rac) declare
(* Elimine, de l'arbre non vide issu de la variable repérée par rac, sa racine *)
   arbre  reference p
do
   connect rac then
      if gauche = nil then rac:= droite else
      if droite = nil then rac:= gauche
      default (* cas général*)
         (* va à gauche du sous-arbre droite*)
         p -> droite;
         until p.gauche = nil repeat
            p -> p.gauche
         repetition;
         (* promeut en rac l'élément p *)
         (rac:=p).gauche := gauche;
         (p.droite =:p).droite := droite
      done (* cas général *)
   done (* connect rac *)
done (* déraciner*)
```

Appliqué à l'arbre de la figure 53, cet algorithme aboutit à celui de la figure 55.

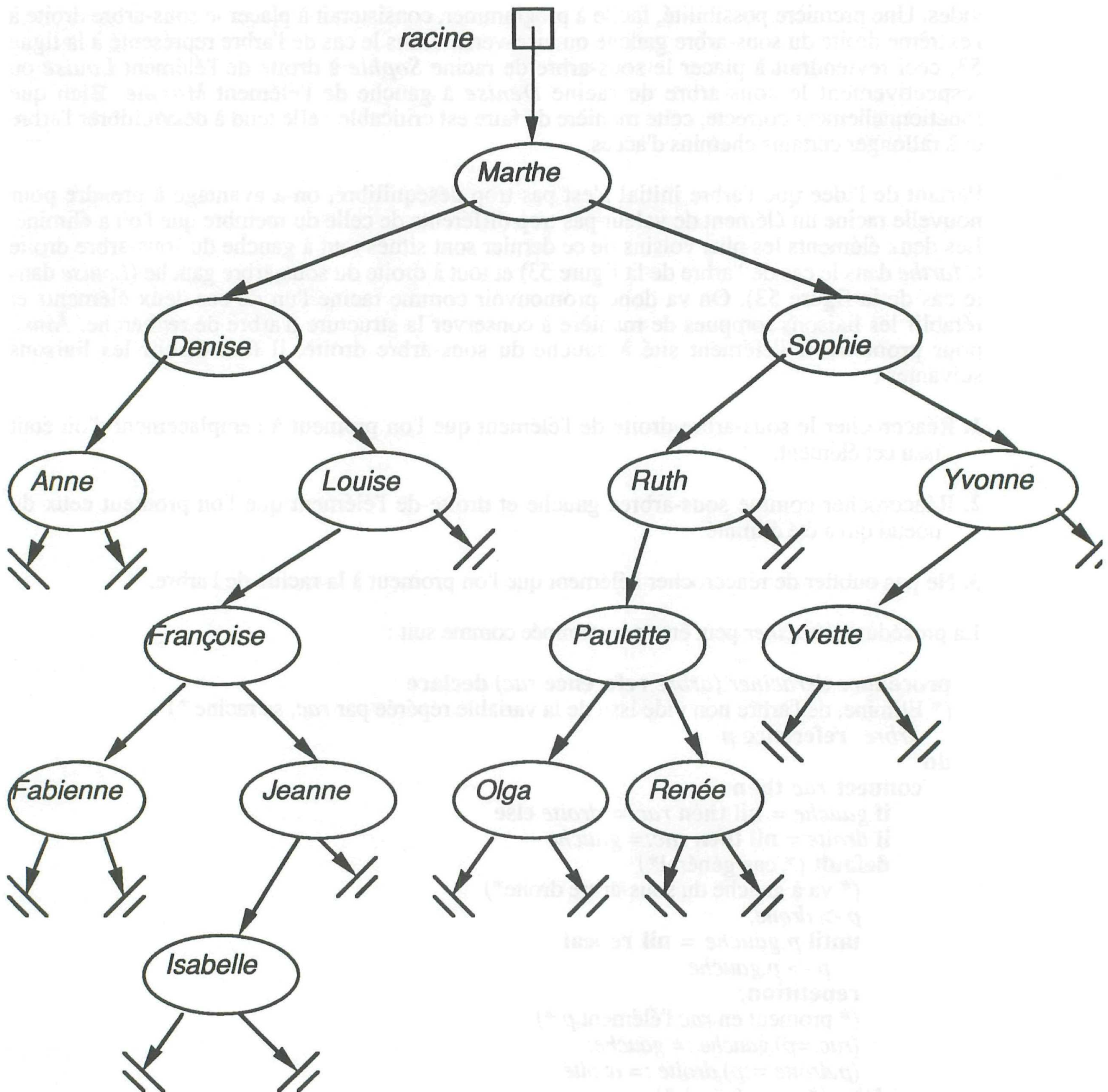


Fig. 55

Cet algorithme destructeur a la propriété désirable de ne rallonger le chemin d'accès à aucun élément et de raccourcir le chemin d'accès à certains éléments.

Soit *a* un arbre de recherche, un itérateur susceptible de traiter, par ordre croissant, les éléments de cet arbre sera conçu comme suit :

1. Si l'arbre *a* est vide, il n'y a (trivialement) rien à faire.

2. Dans le cas général d'un arbre *a* non vide, on procède aux actions successives suivantes :

2.1 Appliquer récursivement le même itérateur au sous-arbre gauche de *a*

2.2 Traiter explicitement l'élément *a.membre*

2.3 Appliquer récursivement le même itérateur au sous-arbre droite de *a*

Soit donc le type procédural :

actor *action (string* **value** *nom)*

On a alors la programmation suivante :

```
procedure parcourir
    (action value acte)
(* Applique l'opération acte [mot ] à chaque élément mot de l'arbre issu de la variable racine;
    les éléments seront traités dans l'ordre alphabétique croissant.
*)
declare
    procedure parc (arbre value rac) do
        connect rac then
            parc (gauche)
            acte [ membre ];
            parc (droite)
        done (* connect rac *)
    done (* parc *)
do parc (racine) done
```

L'application récursive *parc (gauche)* n'intervient pas à la fin du schéma d'exécution de la procédure *parc*. Cette procédure ne satisfait pas aux conditions du théorème de dérécursification; on peut se rendre compte qu'il n'est possible d'en éliminer la récursion qu'en faisant un usage explicite de piles. Appliqué à un arbre de forme arbitraire, cet itérateur consommera un temps proportionnel au nombre d'éléments de l'arbre.

Un arbre de recherche peut être copié au moyen de la fonction récursive *copie* suivante :

```
arbre function copie
    (arbre value rac)
(*Crée une copie conforme de l'arbre rac *)
do (* copie *) take
    connect rac then
        arbre (membre, copie (gauche), copie (droite))
    default nil done
done (* copie *)
```

L'arbre engendré par cet algorithme aura strictement la même forme que l'arbre donné *rac*. On a cependant indiqué qu'il était souhaitable qu'un arbre de recherche soit équilibré. Sans que cela coûte beaucoup plus cher, il est possible de profiter de l'opération de copiage pour engendrer un arbre, fonctionnellement équivalent à l'arbre donné *rac*, mais de forme équilibrée. On suppose pour celà que l'on connaît le nombre d'éléments de l'arbre que l'on veut copier; ce nombre pourra, par exemple, être entretenu dans une variable de la forme :

naturel **variable** *compte*

A défaut d'une telle variable, il peut être obtenu au moyen d'une application appropriée de l'itérateur.

L'arbre initial *rac* et l'arbre équilibré que l'on désire engendrer auront en général des formes complètement différentes. Afin d'éviter une discussion de cas compliquée, on peut avoir recours à une coroutine chargée d'engendrer, par ordre croissant, les éléments de l'arbre.

```
coroutine générateur
    attribute prochain
```

```
(* Engendre, dans l'ordre alphabétique croissant, les membres de l'arbre de recherche issu
   de la variable racine
*)
declare (* générateur*)
   string variable mot ;
   string expression prochain =
   (* Résulte dans le prochain membre de l'arbre concerné

      Condition d'emploi : state générateur = detached
   *)
         (activate générateur now; mot)
do (* générateur *)
   parcourir
      (body action (string value membre) do
         return mot := membre
      done)
done (* générateur *)
```

On constate le truc consistant à utiliser l'itérateur *parcourir* tout en faisant détacher la couroutine *générateur* chaque fois que cet itérateur est prêt à livrer une valeur.

Pour le reste, on construit une fonction récursive *génère_arbre*; appliquée à une valeur entière non-négative *quant*, cette fonction construira un arbre équilibré dans lequel seront incluses les *quant* prochaines valeurs produites par le générateur *prochain*. Le principe en est simple :

1. Si *quant = 0*, il suffit de retourner l'arbre vide **nil**

2. Dans le cas général *quant > 0*, on calcule au préalable le nombre d'éléments de chacun des deux sous-arbres de l'arbre que l'on veut créer. En fait, on peut se rendre compte qu'il faut en attribuer *quant % 2* à l'un et **pred** *quant - quant % 2* à l'autre : ces deux quantités sont égales si *quant* est impair; elles diffèrent d'une unité si *quant* est pair. On procède donc comme suit :

2.1 Par une application récursive *génère_arbre (quant % 2)*, former le sous-arbre gauche *arbre_gauche* de l'arbre que l'on désire créer.

2.2 Ce sous-arbre étant formé, créer un objet du type *arbre* dont le champ *membre* sera initialisé au moyen d'une application de *prochain*, le champ *gauche* au moyen de l'arbre *arbre_gauche* et le champ *droite* au moyen d'une application récursive *génère_arbre* (**pred** *quant - quant % 2)*

Ceci donne l'algorithme suivant :

```
arbre function génère_arbre
   (naturel value quant)
(* le résultat est un arbre de recherche équilibré contenant comme membres les valeurs
   résultant des quant prochaines applications de la fonction prochain
*)
do take
   if quant > 0 then take
      declare
         naturel value taille_gauche = quant % 2;
         arbre value arbre_gauche = génère_arbre (taille_gauche)
      do take
         arbre (prochain,
               arbre_gauche,
               génère_arbre (pred quant-taille_gauche))
      done
```

default nil done
done *(* génère_arbre *)*

La figure 56 indique la structure de la fonction génératrice *table_arborescente.*

```
ville_table expression table_arborescente

    object arbre

    ville_table function crée_table
        (arbre variable racine; naturel variable nombre)
```

fig. 56

En fait, *crée_table* est la "vraie" fonction génératrice; la fonction *table_arborescente* livre simplement comme résultat l'objet *crée_table* (**nil**, *0).*

Le recours à la fonction intermédiaire *crée_table* est nécessaire à l'implantation de la primitive de copiage; cette fonction a la structure suivante (figure 57) :

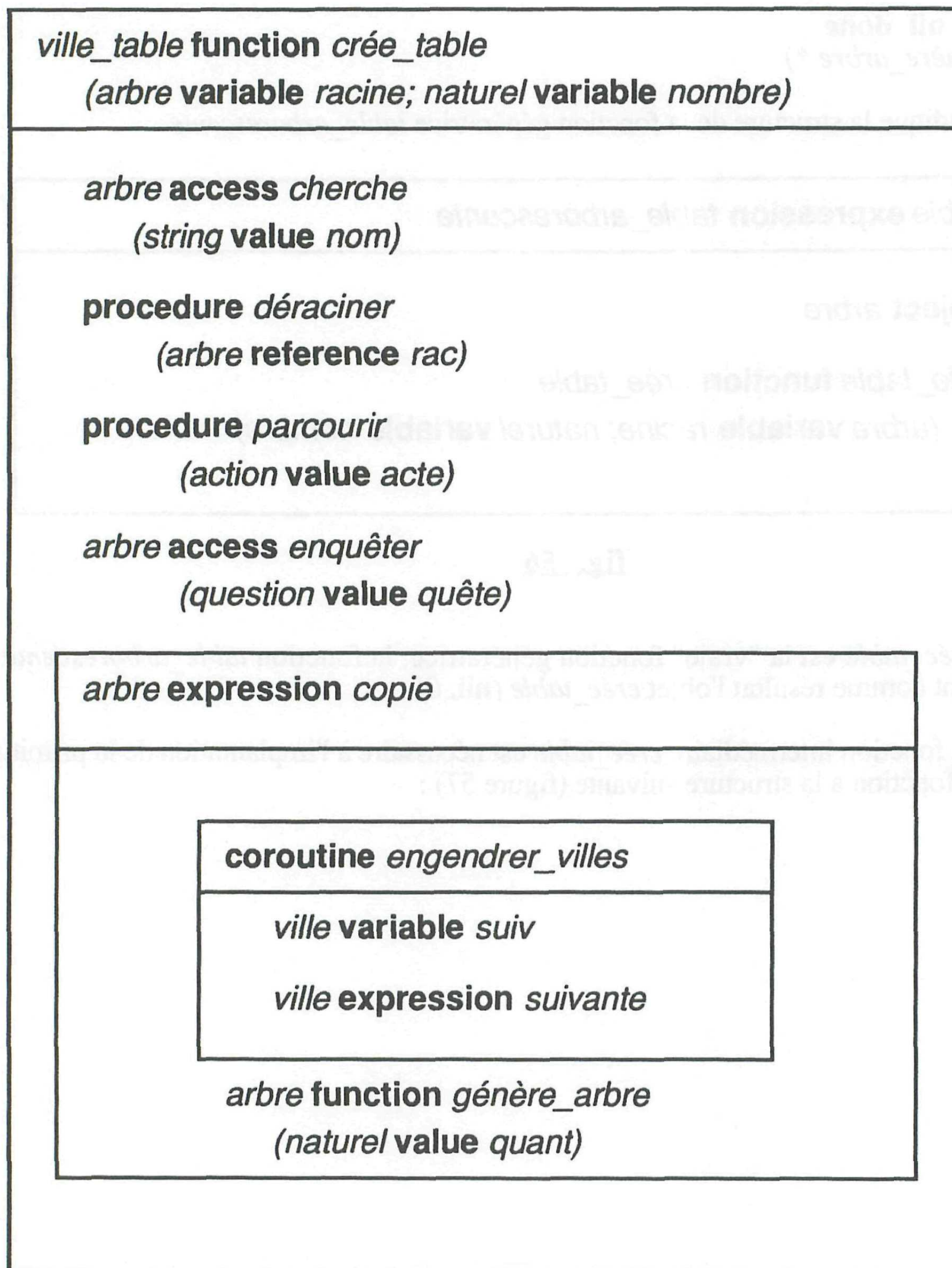

ville_table **function** *crée_table*
(arbre **variable** *racine; naturel* **variable** *nombre)*
arbre **access** *cherche*
(string **value** *nom)*
procedure *déraciner*
(arbre **reference** *rac)*
procedure *parcourir*
(action **value** *acte)*
arbre **access** *enquêter*
(question **value** *quête)*
arbre **expression** *copie*
coroutine *engendrer_villes*
ville **variable** *suiv*
ville **expression** *suivante*
arbre **function** *génère_arbre*
(naturel **value** *quant)*

Fig. 57

```
ville_table EXPRESSION table_arborescente=
(*le resultat est une table associative, initialement vide,
  de villes
*)
  DECLARE(*table_arborescente*)
    OBJECT arbre
      (ville VALUE membre; arbre VARIABLE gauche,droite);

    ville_table FUNCTION cree_table
      (arbre VARIABLE racine; naturel VARIABLE nombre)
    (*cree table dont la representation interne est issue de
      la variable  racine  et qui comporte  nombre  elements;
      les parametres effectifs correspondants devront etre
      consistants
    *)
    DECLARE(*cree_table*)
      arbre ACCESS cherche
        (string VALUE nom)
      DECLARE(*cherche*)
        arbre REFERENCE rac->racine;
        string VALUE nom_maj=UPCASE nom
      DO(*cherche*)
        CYCLE descente DO

          CONNECT rac THEN

          IF nom_maj<UPCASE membre.nom THEN
            rac->gauche
          REPEAT descente DONE;

          IF nom_maj>UPCASE membre.nom THEN
            rac->droite
          REPEAT descente DONE

          DONE(*CONNECT rac*)

        DONE(*descente*)
      TAKE rac DONE(*cherche*);

      PROCEDURE deraciner
        (arbre REFERENCE rac)
      DECLARE arbre REFERENCE p DO
        CONNECT rac THEN
          IF gauche=NIL THEN rac:=droite ELSE
          IF droite=NIL THEN rac:=gauche
          DEFAULT(*cas general: gauche~=NIL/\droite~=NIL*)
            (*va jusu'a gauche du sous-arbre droite*)
            p->droite;
            UNTIL p.gauche=NIL REPEAT
              p->p.gauche
            REPETITION;
          (*promeut en  rac  l'element  p  a gauche du sous-arbre
```

```
 924                  droite
 924                *)
 924                (rac:=p).gauche:=gauche;
 934                (p.droite=:p).droite:=droite
 945              DONE(*cas general*)
 946            DONE(*CONNECT rac*)
 947          DONE(*deraciner*);
 949
 949          PROCEDURE parcourir
 951            (action VALUE acte)
 956          DECLARE(*parcourir*)
 957            PROCEDURE parc(arbre VALUE rac) DO
 965              CONNECT rac THEN
 968                parc(gauche); acte[membre]; parc(droite)
 982              DONE
 983            DONE(*parc*)
 984          DO parc(racine) DONE;
 991
 991          arbre ACCESS enqueter
 994            (question VALUE quete)
 999          DECLARE(*enqueter*)
1000            arbre ACCESS enqu
1003              (arbre REFERENCE rac)
1008            DECLARE
1009              arbre REFERENCE g
1012            DO(*enqu*)TAKE
1014              CONNECT rac THEN TAKE
1018                IF quete[membre] THEN rac ELSE
1026                IF (g->enqu(gauche))~=NIL THEN g
1039                DEFAULT enqu(droite) DONE
1045              DEFAULT rac DONE
1048            DONE(*enqu*)
1049          DO TAKE enqu(racine) DONE;
1057
1057          arbre EXPRESSION copie=
1061            DECLARE(*fabrique une copie*)
1062
1062            COROUTINE engendrer_villes
1064              ATTRIBUTE suivante
1066            (*cette coroutine produit, une a une, dans l'ordre
1066              alphabetique de leurs noms, les villes de la table
1066            *)
1066            DECLARE
1067              ville VARIABLE suiv;
1071
1071              ville EXPRESSION suivante=
1075                (ACTIVATE engendrer_villes NOW; suiv)
1082            DO(*engendrer_villes*)
1083              parcourir
1084                (BODY action(ville VALUE localite) DO
1093                   RETURN suiv:=localite
1097                 DONE)
```

```
         DONE(*engendrer_villes*);

           arbre FUNCTION genere_arbre
             (naturel VALUE quant)
           (*le resultat est un arbre de recherche balance compor=
           tant les  quant  prochaines villes de l'arbre considere
           *)
           DO TAKE
             IF quant>0 THEN TAKE
               DECLARE
                 naturel VALUE taille_gauche=quant%2;
              arbre VALUE arbre_gauche=genere_arbre(taille_gauche)
               DO TAKE
                 arbre(suivante,
                       arbre_gauche,
                       genere_arbre(PRED quant-taille_gauche))
               DONE
             DEFAULT NIL DONE
           DONE(*genere_arbre*)
         DO TAKE genere_arbre(nombre) DONE
     DO(*cree_table*)TAKE
       ville_table
         (cree_table(copie,nombre),
          BODY interrogation DO TAKE nombre=0 DONE,
          BODY quantifieur DO TAKE nombre DONE,
          BODY
            interrogateur(string VALUE nom)
          DO TAKE cherche(nom)~=NIL DONE,
          BODY
            action(ville VALUE localite)
          DECLARE
            arbre REFERENCE ref_noeud->cherche(localite.nom)
          DO
            IF ref_noeud=NIL THEN
              ref_noeud:=arbre(localite,NIL,NIL);
              nombre:=SUCC nombre
            DONE
          DONE,
          BODY
            selecteur(string VALUE nom)
          DECLARE arbre REFERENCE ref_noeud->cherche(nom) DO
            IF ref_noeud=NIL THEN
              ref_noeud:=arbre(ville(nom),NIL,NIL);
              nombre:=SUCC nombre
            DONE
          TAKE ref_noeud.membre DONE,
          BODY
            destructeur(string VALUE nom)
          DECLARE arbre REFERENCE membre->cherche(nom) DO
            UNLESS membre=NIL THEN
              deraciner(membre); nombre:=PRED nombre
            DONE
```

```
            DONE,
            iterateur(parcourir),
            BODY
              enquete(question VALUE quete)
            DO TAKE enqueter(quete)~=NIL DONE,
            BODY
              enquete(question VALUE quete)
            DO TAKE
              enqueter
                (BODY
                   question(ville VALUE localite)
                 DO TAKE ~quete[localite] DONE)
              =NIL
            DONE,
            BODY
              clause_gardee(question VALUE quete; action VALUE acte)
            DECLARE arbre VALUE elem=enqueter(quete) DO
              UNLESS elem=NIL THEN acte[elem.membre] DONE
            DONE,
            BODY
              clause_gardee(question VALUE quete; action VALUE acte)
            DO
              parcourir
                (BODY action(ville VALUE loc) DO
                   IF quete[loc] THEN acte[loc] DONE
                 DONE)
            DONE,
            BODY
              clause_gardee(question VALUE quete; action VALUE acte)
            DECLARE arbre REFERENCE rac DO
              WITHIN rac->enqueter(quete) REPEAT
                acte[membre]; deraciner(rac); nombre:=PRED nombre
              REPETITION
            DONE)
       DONE(*cree_table*)
     DO(*table_arborescente*)TAKE
       cree_table(NIL,0)
     DONE(*table_arborescente*);
 /* /*EJECT*/ */
```

Si l'on tente d'implanter directement la représentation interne dans la partie déclarative de la fonction *table_arborescente*, on ne sait plus comment livrer l'expression associée au paramètre formel *copie* de la classe *ville_table*. Il est évidemment possible de faire appel à la fonction *copie* (telle qu'elle est définie dans *crée_table*) pour créer l'arbre représentatif de la copie; il faudrait ensuite créer une nouvelle instance d'objet du type *ville_table* au moyen d'une application récursive de *table_arborescente* dont on assignerait la copie à la variable *racine*. Cette dernière opération est impossible dès le moment où *table_arborescente* est une fonction et non un objet ou une classe dont on pourrait exporter comme attribut certaines entités. Par contre, depuis l'intérieur de *crée_table*, il est facile d'engendrer la copie souhaitée au moyen d'une application récursive de la forme *crée_table (copie, nombre)*.

On remarque encore la fonction d'accès *enquêter* chargée de trouver un élément *loc* satisfaisant à une condition donnée *quête [loc]*; cette fonction est notamment utilisée pour l'implantation des opérations sélectives *pour_une* et *éliminer_celles*, ainsi des des quantificateurs *pour_toutes* et *il_existe*. Comme n'importe quel élément approprié fait l'affaire, on commence par examiner celui à la racine, à défaut on enquête récursivement dans le sous-arbre gauche puis, le cas échéant, dans le sous-arbre droite. C'est la réalisation du destructeur sélectif qui a imposé le choix d'une fonction d'accès et non d'une fonction ordinaire.

Le recours à des directoires est surtout itéressant pour l'implantation de tables associatives sur mémoires externe (fichiers à accès direct, fichiers indexés séquentiels). Le transfert d'un paquet d'informations entre la mémoire centrale de l'ordinateur est une opération lente relativement au traitement en mémoire centrale.

Il y a donc tout intérêt à concentrer l'information en paquets susceptibles d'être transférés en une fois en mémoire centrale même si le traitement des paquets individuels exige des algorithmes de nature plus complexe. Le recours à des directoires répond à ce souci. Les directoires implantés au moyen de la fonction génératrice *table_directrice* sont en fait gérés entièrement en mémoire centrale; ils illustrent cependant une technique d'implantation possible de fichiers à accès direct. Un directoire peut être considéré comme une sorte d'arbre de recherche généralisé. Etant donnée une valeur entière positive *taille*, un directoire contiendra *taille* membres de la table *membres [1], membres [2],... membres [taille]* ainsi que *taille + 1* liens à des sous-directoires *sous_directoire [0], sous_directoire [1] ... sous_directoire [taille]* (éventuellement vides). Dans le cas d'une implantation sur mémoire périphérique, il faut imaginer que la valeur *taille* soit choisie de manière telle que les *taille* éléments et *taille + 1* liens soient concentrables en un paquet d'information susceptible d'être transféré en une fois du périphérique à la mémoire centrale. Afin d'avoir une structure en arbre de recherche généralisé, on va imposer que les membres du sous-directoire *sous_directoire [k-1]* soient inférieurs à l'élément *membre [k]* et que ceux inclus en *sous_directoire [k]* lui soient supérieurs, ceci pour *k = 1, 2, ... taille;* ceci implique évidemment que les membres d'un directoire seront triés par ordre croissant. La figure 58 en montre un exemple avec *taille = 3*; un tel directoire a une structure d'arbre de rang *taille + 1*; si *taille = 1*, on retrouve un arbre de recherche "traditionnel". Il va de soi que certains directoires peuvent n'être que partiellement remplis; on va supposer pour cela que l'on entretient, dans chaque directoire un compteur *compte* indiquant le nombre de membres de ce directoire (*0 <= compte /\ compte <= taille*); seul un directoire complet comportera des sous-directoires : *compte < taille | > sous_directoire [k]=* **nil**, *k = 1, 2, ... taille*. Un directoire avec compte = *0* ne sera admis qu'au niveau du directoire principal.

Fig. 58

La fonction génératrice *table_directrice* a la structure donnée à la figure 59. La fonction *crée_table* est la "vraie" fonction génératrice; elle est introduite pour l'implantation de l'opération de copie. Les opérations nécessaires à la gestion d'un directoire sont exportées comme attributs des objets de la classe *directoire* dans laquelle est incluse la représentation interne envisagée (Fig. 60).

ville_table **function** *table_directrice* *(positif* **value** *taille)*
class *directoire* *ville_table* **function** *crée_table* *(directoire* **value** *directoire_principal)*

Fig. 59

La fonction *quantité* retourne le nombre d'éléments dans le directoire, y compris ses sous-directoires. La procédure *cherche* est importante; par un procédé de bissections, elle recherche le mot donné dans le directoire considéré. Elle a pour effet de stocker dans les variables associées aux repères *haut* et *bas* des indices tels que l'élément requis sera, le cas échéant, stocké en *membres [bas];* si l'élément ne se trouve pas dans le directoire, sa valeur sera incluse entre *membres [haut]* et *membres [bas]* et il faudrait le chercher dans le sous-directoire *sous_directoire [haut]* : il faut prendre garde que l'on sort de cette procédure avec *bas=haut + 1*, la terminologie *bas, haut* n'étant valable que pendant le déroulement de l'algorithme de recherche. Pour cette raison, la terminologie est inversée lors des applications de *cherche*; ceux-ci ont typiquement la forme *cherche (***upcase** *mot, bas, haut)* : on sort de la procédure avec *haut=bas + 1* et on cherche l'élément requis en *membres [haut]*; s'il ne s'y trouve pas, on passe au sous-directoire *sous_directoire [bas]*. On trouve notamment ce schéma dans la fonction interrogatrice *contient*.

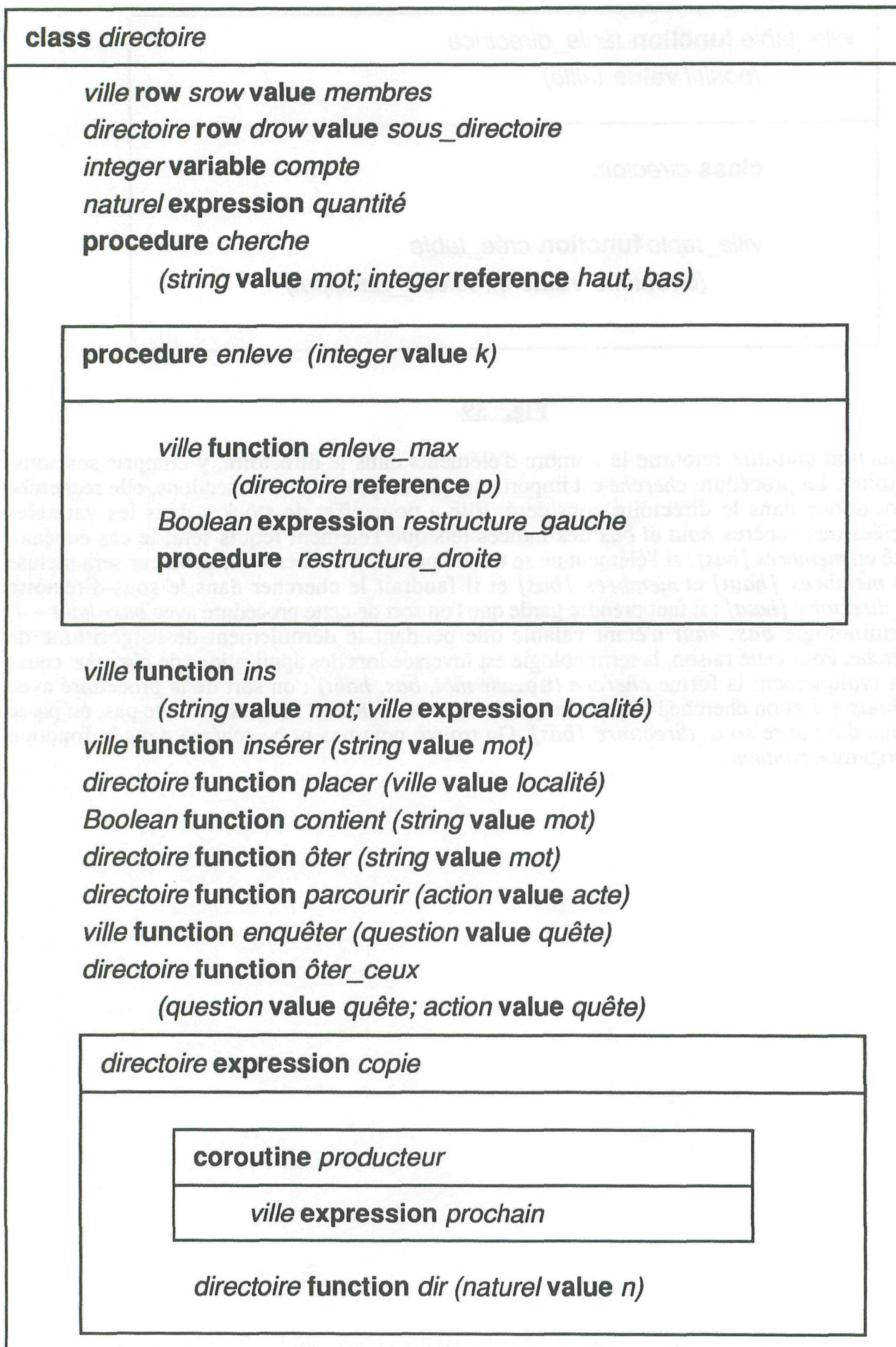

class directoire
ville row srow value membres
directoire row drow value sous_directoire
integer variable compte
naturel expression quantité
procedure cherche
(string value mot; integer reference haut, bas)
procedure enleve (integer value k)
ville function enleve_max
(directoire reference p)
Boolean expression restructure_gauche
procedure restructure_droite
ville function ins
(string value mot; ville expression localité)
ville function insérer (string value mot)
directoire function placer (ville value localité)
Boolean function contient (string value mot)
directoire function ôter (string value mot)
directoire function parcourir (action value acte)
ville function enquêter (question value quête)
directoire function ôter_ceux
(question value quête; action value quête)
directoire expression copie
coroutine producteur
ville expression prochain
directoire function dir (naturel value n)

Fig. 60

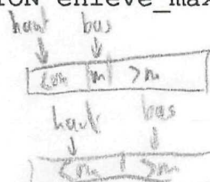
L'algorithme constructeur est incorporé dans la fonction *ins*. Tant que le directoire concerné n'est pas plein, l'insertion a lieu dans ce dernier après recherche et ripage vers le haut des éléments de valeurs supérieures à la sienne; cette dernière opération peut être un peu lente si la valeur de *taille* est élevée : c'est là un des prix à payer de l'applatissement de l'arbre. Si le directoire est plein, l'insertion a lieu (récursivement) dans le sous-directoire approprié, après création de ce dernier s'il n'existe pas déjà. Les fonctions *insérer* et *placer* font appel à ce constructeur; la première est utilisée s'il faut créer un objet nouveau du type *ville* lorsqu'un élément du nom approprié ne figure pas dans le directoire tandis que la seconde place une ville existante dans ce dernier (ceci intervient lors d'un copiage).

Comme dans le cas de l'arbre de recherche, l'algorithme destructeur *ôter* est plus délicat à programmer que le constructeur : il présente plus de cas d'espèce. Après recherche (éventuellement récursive) de l'élément que l'on veut éliminer, ce dernier est retiré au moyen d'une application de la procédure *enleve*; il est fourni à cette dernière en paramètre l'indice, dans la rangée *membres*, de l'élément à purger.Cette opération nécessite en général une restructuration. Si le directoire concerné est plein, on commence par examiner s'il est possible de faire cette restructuration à gauche de l'élément maximum d'un des sous-directoires qui précède celui-ci; plus spécifiquement, si l'on veut supprimer *membres [k]*, on considère successivement *sous_directoire [k-1]*, *sous_directoire [k-2]*,... jusqu'à ce que l'on trouve un sous-directoire d'indice *j* avec *sous_directoire [j]* ~= **nil** : on rippe alors d'une position vers le haut les éléments inclus entre *membres [j + 1]* et *membres [k - 1]* et l'on promeut en *membres [j+1]* l'élément maximum de *sous_directoires [j]*. Si une telle restructuration à gauche s'avère impossible (ce qui sera toujours le cas si le directoire n'est pas plein), on fait une restructuration à droite de l'élément éliminé. Plus spécifiquement, si *membres [k]* est cet élément, on considère successivement les sous-directoires *sous_directoire [k]*, *sous_directoire [k+1]*,... *sous_directoire[compte]*, jusqu'à ce que l'on trouve un sous-directoire *sous_directoire [j]* ~= **nil** en rippant au passage d'une position vers le bas les éléments *membres [k+1]*, *membres [k+2]*,... *membres [j]*; le sous-directoire non vide *sous_directoire [j]* est également rippé en *sous_directoire [j-1]* et l'on promeut son élément maximum en *membres [j]*. Bien entendu, si tous les sous-directoires à droite de *membres [k]* sont vides, on ne fait que ripper d'une position vers le bas les éléments *membres [k+1]* à *membres [compte]* et l'on diminue *compte* d'une unité.

Les autres algorithmes sont d'un abord plus facile et ne seront pas commentés en détail. Dans la fonction *enquêter*, on commence évidemment par chercher dans le directoire concerné si l'un de ses membres satisfait à la requête *quête* donnée avant de passer aux sous-directoires. Le destructeur sélectif *ôter_ceux* procède de bas en haut et de droite à gauche afin de tenter de minimiser les restructurations récursives profondes. Comme dans le cas des arbres de recherche, la fonction *copie* produit une copie équilibrée du directoire concerné en s'appuyant sur une coroutine génératrice *producteur* chargée de produire, dans l'ordre croissant, les membres du directoire (y compris ses sous-directoires) et une fonction récursive *dir* chargée de structurer, en un directoire de profondeur minimum, les éléments engendrés par les *n* prochaines applications de la fonction productrice *prochain*.

```
 1506    ville_table FUNCTION table_directrice
 1509      (positif VALUE taille)
 1514    (*Le resultat est une table associative, initialement vide,
 1514    de villes; l'implantation a lieu au moyen de directoires
 1514      de  taille  villes
 1514    *)
 1514    DECLARE(*table_directrice*)
 1515      CLASS directoire VALUE moi
 1519        INDEX contient
 1521        ATTRIBUTE copie,
 1524          quantite,contient,
 1528          inserer,placer,oter,
 1534          parcourir,enqueter,oter_ceux
 1539      (*Un objet du type  directoire  est cree pour chaque directoire
 1539        et sous-directoire. Il represente un noeud dans un arbre de
 1539        recherche generalise d'ordre  SUCC taille
 1539      *)
 1539      DECLARE(*directoire*)
 1540        ville ROW srow VALUE membres=srow(1 TO taille);
 1553        directoire ROW drow VALUE sous_directoire=
 1559          THROUGH drow(0 TO taille):=NIL REPETITION;
 1570        integer VARIABLE compte:=0;
 1576
 1576        naturel EXPRESSION quantite=
 1580        (*Le nombre d'elements dans le directoire, y-compris ses
 1580          sous-directoires
 1580        *)
 1580          DECLARE integer VARIABLE k:=compte DO
 1587            UNLESS compte<taille THEN
 1592              THROUGH sous_directoire VALUE f REPEAT
 1597                CONNECT f THEN k:=k+quantite  DONE
 1606              REPETITION
 1607            DONE
 1608          TAKE k DONE;
 1612
 1612        PROCEDURE cherche
 1614          (string VALUE mot; integer REFERENCE haut,bas)
 1625        DECLARE integer VARIABLE mid DO
 1630          bas:=1; haut:=compte;
 1638          WHILE bas<=haut REPEAT
 1643            IF
 1644              mot<=UPCASE membres[(mid:=(bas+haut)%2)].nom
 1663            THEN
 1664              haut:=PRED mid
 1668            DEFAULT
 1669              bas:=SUCC mid
 1673            DONE
 1674          REPETITION
 1675        DONE(*cherche*);
 1677
 1677        PROCEDURE enleve(integer VALUE k) DECLARE
 1685          ville FUNCTION enleve_max
```

cas: elt m ds dir. principal



état final

. elt m pas ds "

```
      (directoire REFERENCE p)
    DO(*enleve_max*)TAKE
      CONNECT p THEN TAKE
        DECLARE
          directoire REFERENCE f->sous_directoire[compte];
          ville VARIABLE r
        DO TAKE
          IF f~=NIL THEN
            enleve_max(f)
          DEFAULT
            r:=membres[compte];
            enleve(compte);
            IF compte=0 THEN p:=NIL DONE
          TAKE r DONE
        DONE(*DECLARE f,r*)
      DONE(*CONNECT p*)
    DONE(*enleve_max*);

    Boolean EXPRESSION restructure_gauche=
      DECLARE
        integer VARIABLE j:=k; directoire REFERENCE f
      DO(*restructure_gauche*)TAKE
        CYCLE examen REPEAT
          IF (f->sous_directoire[PRED j])~=NIL THEN
            FOR integer VALUE i FROM k BY -1 TO SUCC j REPEAT
              membres[i]:=membres[PRED i]
            REPETITION;
            membres[j]:=enleve_max(f)
          EXIT examen TAKE TRUE DONE;

          UNLESS (j:=PRED j)>0 EXIT examen TAKE FALSE DONE
        REPETITION
      DONE(*restructure_gauche*);

    PROCEDURE restructure_droite DECLARE
      integer VARIABLE j:=k; directoire REFERENCE f
    DO(*restructure_droite*)
      CYCLE examen REPEAT
        IF (f->sous_directoire[j])~=NIL THEN
          membres[j]:=enleve_max(f);
          directoire[NIL]=:f=:(f->sous_directoire[PRED j])
        EXIT examen DONE;

        UNLESS j<compte THEN
          membres[(PRED compte=:compte)]:=NIL
        EXIT examen DONE;

        membres[j]:=membres[(j:=SUCC j)]
      REPETITION
    DONE(*restructure_droite*)
  DO(*enleve*)
    IF TAKE
```

```
            UNLESS compte<taille THEN
              ~restructure_gauche
            DEFAULT TRUE DONE
          THEN restructure_droite DONE
        DONE(*enleve*);

        ville FUNCTION ins
          (string VALUE mot; ville EXPRESSION localite)
        (*insere dans le directoire une ville de nom  mot  obtenue
          par l'application de l'expression localite . Sans-effet
          si le directoire contient deja une ville du nom donne
        *)
        DECLARE(*ins*)
          string VALUE mot_maj=UPCASE mot;
          integer VARIABLE bas,haut
        DO(*ins*)
          cherche(mot_maj,bas,haut)
        TAKE
          IF TAKE
            IF haut<=compte THEN
              mot_maj~=UPCASE membres[haut].nom
            DEFAULT TRUE DONE
          THEN TAKE
            IF compte<taille THEN
              FOR integer VALUE k
                FROM SUCC compte=:compte BY -1 TO haut
              REPEAT membres[SUCC k]:=membres[k] REPETITION
            TAKE
              (membres[haut]:=localite)
            DEFAULT
              IF sous_directoire[bas]=NIL THEN
                sous_directoire[bas]:=directoire
              DONE
            TAKE
              sous_directoire[bas].ins(mot,localite)
            DONE
          DEFAULT membres[haut] DONE
        DONE(*ins*);

        ville FUNCTION inserer
          (string VALUE mot)
        (*Insere dans le directoire un nouvel element de valeur mot ;
         sans effet si le directoire possede deja une telle composan-
          te. Resulte dans la ville concernee.
        *)
        DO TAKE ins(mot,ville(mot)) DONE;

        directoire FUNCTION placer
          (ville VALUE localite)
        (*Insere, dans la directoire considere, la ville  localite ;
          sans effet si le directoire contient deja cette ville. Le
          resultat est le directoire concerne  moi
```

```
2115         *)
2115         DO ins(localite.nom,localite) TAKE moi DONE;
2128
2128         Boolean FUNCTION contient
2131           (string VALUE mot)
2136         (*TRUE ssi le directoire concerne contient une composante de
2136           valeur  mot .
2136         *)
2136         DECLARE(*contient*)
2137           string VALUE mot_maj=UPCASE mot;
2144           integer VARIABLE bas,haut
2149         DO
2150           cherche(mot_maj,bas,haut)
2158         TAKE
2159           IF TAKE
2161             IF haut<=compte THEN
2166               mot_maj=UPCASE membres[haut].nom
2175             DEFAULT FALSE DONE
2178           THEN TRUE ELSE
2181           CONNECT sous_directoire[bas] THEN
2187             contient(mot)
2191           DEFAULT FALSE DONE
2194         DONE(*contient*);
2196
2196         directoire FUNCTION oter
2199           (string VALUE mot)
2204         (*Elimine du directoire concerne sa composante de valeur
2204           mot ;  sans effet si le directoire ne possede aucune
2204           telle composante. Resulte dans le directoire concerne
2204           moi .
2204         *)
2204         DECLARE
2205           string VALUE mot_maj=UPCASE mot;
2212           integer VARIABLE bas,haut;
2218           directoire REFERENCE f
2221         DO(*oter*)
2222           cherche(mot_maj,bas,haut);
2231           IF TAKE
2233             IF haut<=compte THEN
2238               mot_maj~=UPCASE membres[haut].nom
2247             DEFAULT TRUE DONE
2250           THEN(*pas trouve; va dans sous-directoire*)
2251             CONNECT f->sous_directoire[bas] THEN
2259               oter(mot);
2264               IF compte=0 THEN f:=NIL DONE
2273             DONE(*CONNECT f*)
2274           DEFAULT enleve(haut) DONE
2280         TAKE moi DONE(*oter*);
2284
2284         directoire FUNCTION parcourir
2287           (action VALUE acte)
2292         (*Parcourt, dans l'ordre lexicographique croissant de leur
```

```
2292        nom, toutes les composantes du directoire concerne; pour
2292        chacune d'entre elles, effectue l'enonce  acte[localite] .
2292        Le resultat est le directoire concerne
2292      *)
2292      DECLARE integer VARIABLE k:=0 DO
2299        WHILE
2300          CONNECT sous_directoire[k] THEN
2306            parcourir(acte)
2310          DONE
2311        TAKE (k:=SUCC k)<=compte REPEAT
2321          acte[membres[k]]
2328        REPETITION
2329      TAKE moi DONE(*parcourir*);
2333
2333      ville FUNCTION enqueter
2336        (question VALUE quete)
2341      (*Cherche si le directoire, ou l'un de ses sous_directoires
2341        possede une ville  localite  pour laquelle quete[localie]
2341        est vraie. En cas de reponse affirmative, le resultat
2341        est une telle localite. Dans le cas contraire, le resultat
2341        est NIL
2341      *)
2341      DECLARE
2342        integer VARIABLE k:=0; ville VARIABLE loc
2351      DO(*enqueter*)TAKE
2353        CYCLE recherche_principale REPEAT
2356
2356        IF (k:=SUCC k)<=compte THEN
2366          IF
2367            quete[membres[k]]
2374          EXIT recherche_principale TAKE
2377            membres[k]
2381          DONE
2382        REPEAT recherche_principale DONE;
2386
2386        (*rien dans le directoire principal: examine les
2386          directoires secondaires
2386        *)
2386          k:=0;
2390          CYCLE recherche_secondaire REPEAT
2393            CONNECT sous_directoire[k] THEN
2399              IF
2400                (loc:=enqueter(quete))~=NIL
2410              EXIT recherche_principale TAKE
2413                loc
2414              DONE
2415            DONE;
2417
2417            IF
2418              (k:=SUCC k)>compte
2426            EXIT recherche_principale TAKE NIL DONE
2431          REPETITION(*recherche_secondaire*)
```

```
        REPETITION(*recherche_principale*)
      DONE(*enqueter*);

      directoire FUNCTION oter_ceux
        (question VALUE quete; action VALUE acte)
      (*elimine, dans un ordre non defini a priori, les villes
        loc  telles que  quete[loc]  est vrai; pour chacune de
        ces villes, effectue l'enonce  acte[loc]. Resulte dans
        le directoire concerne  moi .
      *)
      DECLARE(*oter_ceux*)
        ville VARIABLE membre; directoire REFERENCE f
      DO(*oter_ceux*)
        (*fait l'elimination de bas en haut et de droite a gauche*)
        FOR integer VALUE k FROM compte BY -1 TO 0 REPEAT
          CONNECT f->sous_directoire[k] THEN
            oter_ceux(quete,acte);
            IF compte=0 THEN f:=NIL DONE
          DONE
        REPETITION;
        FOR integer VALUE k FROM compte BY -1 TO 1 REPEAT
          WHILE TAKE
            CONNECT membre:=membres[k] THEN
              quete[membre]
            DEFAULT FALSE DONE
          REPEAT enleve(k); acte[membre] REPETITION
        REPETITION
      TAKE moi DONE(*oter_ceux*);

    directoire EXPRESSION copie=
    (*Resulte dans un nouveau directoire contenant les memes
      villes que le directoire concerne  moi .
    *)
      DECLARE
        COROUTINE producteur  ATTRIBUTE prochain  DECLARE
          ville VARIABLE loc;
          ville EXPRESSION prochain=
            (ACTIVATE producteur NOW; loc)
        DO(*producteur*)
          parcourir
            (BODY action(ville VALUE localite) DO
               RETURN loc:=localite
             DONE)
        DONE(*producteur*);

        directoire FUNCTION dir
          (naturel VALUE n)
        DO(*dir*)TAKE
          IF n=0 THEN NIL ELSE
          IF n<=taille THEN TAKE
            DECLARE directoire VALUE d=directoire DO
              FOR integer VALUE k FROM 1 TO n REPEAT
```

```
                 d.membres[k]:=prochain
               REPETITION;
               d.compte:=n
             TAKE d DONE
           DEFAULT(*n>taille; il y aura des sous-directoires*)TAKE
             DECLARE directoire VALUE d=directoire;
               integer VARIABLE taille_fils:=n%(SUCC taille);
               integer VALUE premier_cadet=SUCC n\(SUCC taille);
               integer VARIABLE k:=0
             DO
               WHILE
                 d.sous_directoire[k]:=dir(taille_fils)
               TAKE k<taille REPEAT
                 d.membres[(k:=SUCC k)]:=prochain;
                 IF k=premier_cadet THEN
                   taille_fils:=PRED taille_fils
                 DONE
               REPETITION;
               d.compte:=taille
             TAKE d DONE
           DONE(*CASE n*)
         DONE(*dir*)
       DO(*copie*)TAKE
         CONNECT dir(quantite) THEN
           moi
         DEFAULT directoire(*vide*) DONE
       DONE(*copie*)
     DO(*directoire*)DONE; (*class directoire*)

     ville_table FUNCTION cree_table
       (directoire VALUE directoire_principal)
     (*le resultat est une table dont la representation interne
       est le directoire donne directoire_principal
     *)
     DO(*directoire_principal*)TAKE
       ville_table
         (cree_table(directoire_principal.copie),
          BODY interrogation DO TAKE
            directoire_principal.quantite=0
          DONE,
          BODY quantifieur DO TAKE
            directoire_principal.quantite
          DONE,
          BODY
            interrogateur(string VALUE mot)
          DO TAKE directoire_principal.contient(mot) DONE,
          BODY
            action(ville VALUE localite)
          DO directoire_principal.placer(localite) DONE,
          BODY
            selecteur(string VALUE nom)
          DO directoire_principal.inserer(nom) DONE,
```

```
villes                                            Vax Newton Compiler 0.2c16
                                                  Page   19
                    Source listing

              BODY
                destructeur(string VALUE nom)
              DO directoire_principal.oter(nom) DONE,
              BODY
                iterateur(action VALUE acte)
              DO directoire_principal.parcourir(acte) DONE,
              BODY
                enquete(question VALUE quete)
              DO TAKE
                directoire_principal.enqueter(quete)~=NIL
              DONE,
              BODY
                enquete(question VALUE quete)
              DO TAKE
                directoire_principal.enqueter
                  (BODY
                      question(ville VALUE localite)
                   DO TAKE ~quete[localite] DONE)
                =NIL
              DONE,
              BODY
                clause_gardee(question VALUE quete; action VALUE acte)
              DECLARE
                ville VALUE loc=directoire_principal.enqueter(quete)
              DO
                UNLESS loc=NIL THEN acte[loc] DONE
              DONE,
              BODY
                clause_gardee(question VALUE quete; action VALUE acte)
              DO
                directoire_principal.parcourir
                  (BODY action(ville VALUE localite) DO
                     IF quete[localite] THEN acte[localite] DONE
                   DONE)
              DONE,
              BODY
                clause_gardee(question VALUE quete; action VALUE acte)
              DO directoire_principal.oter_ceux(quete,acte) DONE)
          DONE(*cree_table*)
        DO(*table_directrice*)TAKE
          cree_table(directoire(*vide*))
        DONE(*table_directrice*);
        /* /*EJECT*/ */
```

Du module *reseau* (fig. 61) sont exportées les deux tables *reseau_principal* et *reseau_secondaire*. La première contient initialement le réseau complet tel qu'il a été enregistré. Après cet enregistrement, une opération de copie restructure le directoire utilisé pour sa représentation de manière à minimiser la profondeur des sous-directoires.

```
module reseau
    constant taille_table
    ville_table variable res_pr value reseau_principal
    ville_table value reseau_secondaire
    string variable texte
    constant alpha
    string expression nom
    real expression nombre
```

Fig. 61

La table *reseau_secondaire*, initialement vide, sera structurée en arbre de recherche. Dans cette version du programme, il est exporté du module les fonctions *nom* et *nombre*, ainsi que la variable *texte* dont il sera fait usage au niveau du programme principal lors de l'enregistrement des localités à partir desquelles on veut obtenir les itinéraires minimaux ainsi que le rayon définissant le voisinage dont on veut connaître les localités.

```
MODULE reseau
  ATTRIBUTE reseau_principal,reseau_secondaire,
            texte,nom,nombre
 (*Enregistre le reseau concerne; chaque troncon de route y est
  presente sous la forme d'une ligne de texte contenant, sous
  la forme de chaines, le nom des deux localites encadrantes
  suivi de sa longueur. La description du reseau sera suivie
  d'une ligne contenant la chaine  <<<FIN>>> .
 *)
DECLARE(*reseau*)
  CONSTANT taille_table=15;

  ville_table VARIABLE res_pr VALUE reseau_principal
    :=table_directrice(taille_table);
  ville_table VALUE reseau_secondaire=table_arborescente;

  string VARIABLE texte;
  (*La derniere ligne lue du fichier de donnees*)

  CONSTANT alpha={FROM "A" TO "Z",FROM "a" TO "z","-","'","_"};
  (*Les caracteres susceptibles d'apparaitre dans les noms de
    ville; le caractere de soulignage y sera systematiquement
    remplace par l'espace blanc.
  *)

  string EXPRESSION nom=
  (*Extrait du debut de  texte  un nom de localite*)
    DECLARE
      string VARIABLE res:=alpha SPAN (texte:=" "-texte)
    DO
      WHILE "_" IN res REPEAT
        res:=res LEFTOF "_"+" "+res LEFTCUT "_"
      REPETITION;
      texte:=alpha-texte
    TAKE res DONE;

  CONSTANT num={FROM "0" TO "9"};

  real EXPRESSION nombre=
  (*Extrait du debut de  texte  une valeur reelle*)
    DECLARE
      string VALUE
        ent=num SPAN (texte:=" "-texte),
        frc=IF "." STARTS (texte:=num-texte) THEN
              texte:=texte LEFTCUT "."
            TAKE
              num SPAN (num-texte=:texte)
            DEFAULT "" DONE;
      real VARIABLE res:=0
    DO
      THROUGH ent+frc VALUE dig REPEAT
        res:=10*res+(ORD dig-ORD "0")
```

```
villes                                           Vax Newton Compiler 0.2c16
                                          Page   21
                 Source listing

      REPETITION
    TAKE res*10**(-LENGTH frc) DONE;
DO(*reseau*)
  print("*****Description du reseau*****",line);
  DECLARE
    naturel VARIABLE numero:=0;
    (*Le numero du dernier troncon*)

    string VARIABLE org,dest; real VARIABLE long
  DO
    UNTIL
      read(texte)
    TAKE " "-texte-" "="<<<FIN>>>" REPEAT
      edit((numero:=SUCC numero),4,0);
      print("--->",texte,line);
      org:=nom; dest:=nom; long:=nombre;
      IF " "-texte="" THEN
        reseau_principal[org].relier
          (numero,long,reseau_principal[dest]);
        reseau_principal[dest].relier
          (numero,long,reseau_principal[org])
      DEFAULT
        print(___"###ligne precedante incorrecte ignoree###",line)
      DONE
    REPETITION
  DONE(*DECLARE numero,... *);
  print("<<<FIN>>>",page);
  res_pr:=reseau_principal.copie
DONE(*reseau*);
/* /*EJECT*/ */
```

Dans la partie exécutable du programme, les localités à partir desquelles les itinéraires doivent être calculés sont successivement enregistrées et traitées. On peut constater que toutes les localités inaccessibles depuis la première d'entre elles seront éliminées du *réseau_principal* pour constituer le *réseau_secondaire*. Même lorsque le réseau secondaire comporte plusieurs sous-réseaux disjoints, les localités inaccessibles depuis celle à partir de laquelle on cherche les itinéraires n'en sont pas purgées,elles ne sont éliminées que de la copie de cette table mise dans la variable *reseau_considéré*.

```
    ville_table VARIABLE reseau_considere;

    string VARIABLE nom_de_ville; real VARIABLE rayon;
    ville VARIABLE origine_parcours;

    question VALUE voisinage=
      BODY
        question(ville VALUE localite)
      DO TAKE
        localite.distance<=rayon EXCL localite=origine_parcours
      DONE
  DO(*villes*)
    UNTIL end_file REPEAT
      read(texte); nom_de_ville:=nom; rayon:=nombre;
      reseau_considere:=
        IF reseau_principal.contient(nom_de_ville) THEN
          reseau_principal ELSE
        IF reseau_secondaire.contient(nom_de_ville) THEN
          reseau_secondaire.copie
        DEFAULT NIL DONE;
      CONNECT reseau_considere THEN
        parcourir
          (BODY action(ville VALUE localite) DO
             localite.init
           DONE);
        (origine_parcours:=reseau_considere[nom_de_ville]).origine;
        UNLESS
          pour_toutes
            (BODY
               question(ville VALUE localite)
             DO TAKE localite.atteinte DONE)
        THEN
          print("---Localites inaccessibles depuis"_,nom_de_ville,
                _"---",line);
          DECLARE string VARIABLE texte:="" DO
            eliminer_celles
              (BODY
                 question(ville VALUE localite)
               DO TAKE ~localite.atteinte DONE,
               BODY
                 action(ville VALUE localite)
               DO
                 texte:=texte+localite.nom;
                 IF LENGTH texte>60 THEN
                   print(texte,line); texte:=""
                 DEFAULT texte:=texte+"  " DONE;
                 IF reseau_considere=reseau_principal THEN
                   reseau_secondaire.inserer(localite)
                 DONE
               DONE);
            UNLESS texte="" THEN print(texte,line) DONE
          DONE(*DECLARE texte*)
```

```
DEFAULT
  print("+++Tout le reseau considere accessible depuis"_,
        nom_de_ville,_"+++",line)
DONE;
line;
print("***Itineraires depuis"_,nom_de_ville,"***",line,line);
parcourir
  (BODY
     action(ville VALUE localite)
   DO
     CONNECT localite THEN
       print(___,nom); column(25);
       IF atteinte THEN
         IF nom=nom_de_ville THEN
           print("***origine du parcours***")
         DEFAULT
           edit(distance,6,1); print("Km. via"_);
           UNLESS etape.nom=nom_de_ville THEN
             print(etape.nom,_"et"_)
           DONE;
           print("route"_); edit(itineraire,3,0)
         DONE
       DEFAULT print("---localite inaccessible---") DONE
     DONE;
     line
   DONE);
line;
IF il_existe(voisinage) THEN
  print("+++Localites dans un rayon de"_,
        edit(rayon,6,1),"Km. de"_,nom_de_ville,"+++",line);
  DECLARE string VARIABLE texte:="" DO
    pour_celles
      (voisinage,
       BODY action(ville VALUE localite) DO
         texte:=texte+localite.nom;
         IF LENGTH texte<60 THEN
           texte:=texte+"  "
         DEFAULT print(texte,line); texte:="" DONE;
       DONE);
    UNLESS texte="" THEN print(texte,line) DONE
  DONE(*DECLARE texte*)
DEFAULT
  print("---Toutes les localites sont a plus de"_,
        edit(rayon,6,1),"Km. de"_,nom_de_ville,
        "---",line)
DONE;
print("<<<FIN>>>",line,line)
DEFAULT
print(___,"###Localite inconnue"_,nom_de_ville,"###",line)
DONE
REPETITION;
print("<<<<<FIN DES APPLICATIONS>>>>>")
```

```
DONE(*villes*)

     ****  No messages were issued  ****
```



t(n) = temps moyen construct° arbre de n élé'ts

t(n,m) = temps moyen construct° arbre de n élé'ts si l'hypothèse que le 1er élé't inséré est le m-ième de l'ordre impliqué par la relat° <.

$$t(n,m) = \alpha + \beta(n-1) + t(m-1) + t(n-m)$$

$$t(n) = \alpha + \beta(n-1) + \frac{1}{n}\sum_{m=1}^{n} t(m-1) + \frac{1}{n}\sum_{m=1}^{n} t(n-m)$$

$$= \alpha + \beta(n-1) + \frac{2}{n}\sum_{j=0}^{n-1} t(j)$$

$$n\,t(n) = \alpha n + \beta n(n-1) + 2\sum_{j=0}^{n-1} t(j) \qquad \Big|\ (-1)$$

$$(n+1)\,t(n+1) = \alpha(n+1) + \beta(n+1)n + 2\sum_{j=0}^{n} t(j) \qquad \Big|\ 1$$

$$(n+1)\,t(n+1) - n\,t(n) = \alpha + 2n\beta + 2t(n)$$

$$(n+1)\left[t(n+1) - t(n)\right] - t(n) = \alpha + 2\beta n \qquad \delta(n) = t(n+1) - t(n)$$

$$(n+1)\,\delta(n+1) - t(n) = \alpha + 2\beta n \qquad \Big|\ 1$$

$$n\,\delta(n) - t(n-1) = \alpha + 2\beta(n-1) \qquad \Big|\ (-1)$$

$$(n+1)\,\delta(n+1) \underbrace{- n\,\delta(n) - \delta(n)}_{-(n+1)\delta(n)} = 2\beta \qquad H(n) = \sum_{j=1}^{n} \frac{1}{j} = O(\ln n)_{n\to\infty}$$

$$\delta(n+1) = \delta(n) + \frac{2\beta}{n+1}$$

$$\delta(n) = \delta(1) + 2\beta\left[\frac{1}{2} + \frac{1}{3} + \dots \frac{1}{n}\right]$$

$$\delta(n) = \alpha + 2\beta\left[H(n) - 1\right] = O(\ln n)_{n\to\infty} \Rightarrow t(n) = 2\alpha + 2\beta\sum_{j=1}^{n}\left[H(j) - 1\right] = O(n\ln n)_{n\to\infty}$$

Chapitre 15

Parallélisme

Toutes les applications vues jusqu'à ce stade étaient de nature séquentielle; en effet, on peut considérer que chaque programme était exécuté par un seul processeur (une seule unité de traitement) qui élaborait dans un ordre bien défini les énoncés de l'application. Une manière d'accélérer le traitement d'une application est de la décomposer en plusieurs parties susceptibles d'être exécutées simultanément sur une installation dotée de plusieurs processeurs; on dira alors que l'application est élaborée en parallèle. Le parallélisme peut être conçu de plusieurs manières; on dispose par exemple de calculatrices vectorielles; une telle calculatrice est dotée de familles de processeurs capables de réaliser en parallèle une opération sur les diverses composantes d'un vecteur. Ainsi, deux vecteurs pourront être ajoutés en parallèles; le calcul des composantes individuelles du vecteur somme seront faits simultanément. Dans un tel cas, chaque processeur d'une telle famille effectuera, à un instant donné, la même opération sur l'une des composantes des vecteurs concernés. Ce genre d'architecture, susceptible d'être très performante, est bien adaptée au traitement de grandes applications de nature matricielle, par exemple des systèmes d'équations résultant de la discrétisation d'équations aux dérivées partielles. Les applications de simulation continue peuvent déboucher sur des systèmes capables d'être traités efficacement sur un ordinateur vectoriel.

D'autres applications ne se laissent pas si commodément vectoriser. Pour les paralléliser, il est souhaitable de disposer de processeurs susceptibles de réaliser simultanément des algorithmes différents. C'est ce cas que l'on va considérer par la suite; plusieurs langages de programmation modernes possèdent des primitives permettant d'exprimer des algorithmes parallèles; c'est en particulier le cas de Algol-68, Portal, Ada et désormais Newton. Les algorithmes parallèles écrits dans un de ces langages peuvent d'ailleurs fort bien être élaborés sur une calculatrice dotée d'un seul processeur; le parallélisme sera alors simulé : si, à un moment donné, plusieurs tâches sont susceptibles d'être exécutées en parallèle, le processeur sera attribué à tour de rôle à chacune de ces tâches pendant un intervalle de temps donné (par exemple *20 ms*) au bout duquel la tâche sera momentanément interrompue et le processeur attribué à la tâche suivante dans la liste (Fig. 62). Il va de soi que dans le cas d'un tel système mono-processeur, il n'y aura aucun gain de temps par rapport à un algorithme équivalent de nature séquentielle (au contraire). Dans un contexte multi-processeur, on pourra garder un contexte d'exécution analogue à celui de la figure 62; cependant des processeurs seront attribués à plusieurs des tâches susceptibles d'être exécutées en parallèle. De nouveau, s'il y a plus de tâches que de processeurs, il faudra permuter circulairement l'attribution des processeurs au bout de tranches de temps fixées à priori.

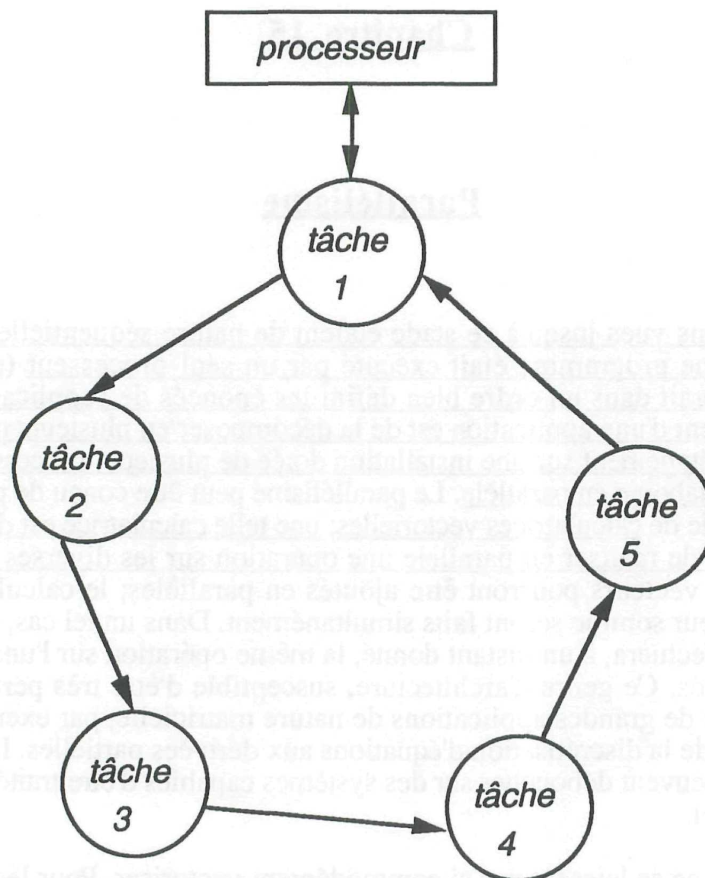


Fig. 62

La programmation parallèle pose des problèmes nouveaux; elle est d'ailleurs d'un ordre de grandeur plus difficile que la programmation séquentielle. Le débogage des programmes est également plus délicat.

Deux problèmes fondamentaux en programmation parallèle sont l'accès à des ressources (par exemple des variables) communes et la synchronisation des tâches parallèles.

Plusieurs tâches peuvent chercher à accéder au même moment à une même variable; on a les trois cas suivants :

- Si deux ou plusieurs tâches cherchent à lire, sans le modifier, le contenu d'une variable, chacune des tâches accèdera à la valeur correcte de la variable.

- Si une tâche cherche à lire le contenu d'une variable tandis qu'une autre tâche tente d'en modifier son contenu, le résultat de l'opération est imprévisible. En particulier, il n'est pas défini si c'est l'ancien ou le nouveau contenu de la variable qui sera obtenu par la première tâche.

- Si deux ou plusieurs tâches cherchent à modifier simultanément le contenu d'une variable, le résultat de l'opération est imprévisible. Il n'est pas défini à priori quel sera le contenu final de la variable.

Dans les deux derniers cas, il sera donc essentiel de séparer dans le temps les accès à la variable concernée. On dira qu'il faut assurer l'exclusion mutuelle des énoncés qui réalisent ces accès.

On peut illustrer ceci dans le cas d'une simple instruction d'assignation; soit la déclaration suivante :

integer **variable** *compte : = 0*

On suppose que deux tâches exécutent au même moment l'énoncé *compte : = compte + 1*; il n'y a aucune garantie que la variable *compte* soit effectivement incrémentée deux fois. Il peut se produire que chacune des deux tâches extraie la (même) valeur primitive de la variable pour évaluer l'expression *compte + 1* et stocker sa valeur dans la variable; s'il en est ainsi, la variable n'aura été incrémentée qu'une seule fois. Pour garantir la double incrémentation, l'assignation *compte : = compte + 1* devra être exécutée en exclusion mutuelle.

Le programme *acces_mutuels* suivant en montre à l'évidence la nécessité.

```
acces_mutuels                                        Vax Newton Compiler 1.0
                                                 Page      1
                     Source listing

      1   /* /*OLDSOURCE=USER2:[RAPIN]ACCES_MUTUELS.NEW*/ */
      1   PROGRAM acces_mutuels DECLARE
      4     CONSTANT nombre_processeurs=1000,nombre_increments=1000;
     13
     13     integer VARIABLE compte:=0;
     19
     19     PROCESSOR incrementeur DO
     22       FOR integer FROM 1 TO nombre_increments REPEAT
     29         compte:=SUCC compte
     33       REPETITION
     34     DONE(*incrementeur*)ROW vec_incr VALUE table_incrementeurs=
     40       THROUGH
     41         vec_incr(1 TO nombre_processeurs):=incrementeur
     49       REPETITION;
     51
     51     Boolean VARIABLE termine
     54   DO(*acces_mutuels*)
     55     UNTIL
     56       termine:=TRUE;
     60       THROUGH
     61         table_incrementeurs VALUE proc
     64       REPEAT
     65         termine:=termine/\STATE proc=finished
     73       REPETITION
     74     TAKE termine REPEAT SWITCH REPETITION;
     80     print("Valeur finale du compteur:"_,compte)
     87   DONE(*acces_mutuels*)

          ****  No messages were issued  ****
```

Deux résultats d'exécution de ce programme.

```
Valeur finale du compteur:        939906
Valeur finale du compteur:        912176
```

Une déclaration de processeur est analogue à une déclaration de classe ou de processus; le mot clé **processor** y remplace **class** ou respectivement **process**. Une déclaration de processeur définit un type d'objet dont l'exécution se déroule, dès sa création, en parallèle avec la tâche qui l'a créé. A priori, une tâche du type *incrémenteur* va augmenter de *nombre_incréments = 1000* la valeur de la variable *compte*.

On constate qu'il est créé une rangée *table_incrémenteurs* de *nombre_processeurs = 1000* tâches *incrémenteur*. Ces tâches sont exécutées, dès leur création, en parallèle avec le programme principal. La partie exécutable du programme principal a pour rôle d'attendre que chacune des

tâches *incrémenteur* ait terminé son exécution et d'imprimer la valeur résultante de *compteur*. Pour examiner si une tâche a achevé son exécution, il est possible de lui appliquer l'opérateur **state**; ceci est donc analogue au traitement des coroutines. On verra d'ailleurs que cette analogie va beaucoup plus loin. La déclaration présumée du type prédéfini *status* (chapitre 7) doit être complétée de la manière suivante :

scalar *status*
(null, attached, detached, terminated,
inexistant, ready, running, waiting, finished, failed)

Les valeurs entre *null* et *terminated* sont produites lorsque l'opérateur **state** est appliqué à une coroutine tandis que les autres le sont lorsqu'il est appliqué à une tâche que l'on peut dénommer paroutine (par analogie à coroutine). Ces dernières valeurs ont les significations suivantes :

state *t* = *inexistant* la tâche *t* est la tâche vide notée **nothing**

ready la tâche *t* fait partie des tâches susceptibles d'être exécutées; au moment considéré, il ne lui est cependant attaché aucun processeur (physique); son exécution pourra être reprise sans autre dans une tranche de temps subséquente.

running la tâche *t* est en cours d'exécution par un processeur (physique).

waiting la tâche *t* n'est pas en cours d'exécution; la reprise de son élaboration devra être explicite.

finished la tâche *t* a achevé, de manière normale, son exécution.

failed la tâche *t* a achevé de manière anormale son exécution.

On note encore la clause **switch**; cette clause force un changement d'attribution des processeurs (un peu comme à la fin d'une tranche de temps). Plus spécifiquement, si le système comporte (au moins) une tâche dans l'état *ready*, le processeur physique attaché à la tâche courante, notée **curr_task** en est détaché pour être attribué à une tâche dans l'état *ready*. L'effet global est donc que la tâche qui exécute la clause **switch** passe à l'état *ready* tandis qu'une des tâches à l'état *ready* passe à l'état *running*. Exécutée lorsqu'aucune tâche n'est dans l'état *ready*, la clause **switch** n'a aucun effet. D'une manière générale, un **switch** peut être utilisé lorsque la tâche courante n'a rien d'utile à faire pour le moment mais que d'autres tâches, avec lesquelles elle est en cours d'exécution parallèle, peuvent rapidement amener un changement susceptible de lui permettre de reprendre son exécution.

A priori, on s'attendrait que l'élaboration du programme *accès_mutuels* produise pour résultat la valeur *nombre_incréments*nombre_processeurs = 1_000_000*; on constate qu'il n'en est rien. Deux exécutions du programme ont d'ailleurs produit des résultats différents; dans les deux cas, ces résultats sont inférieurs à la valeur attendue. Le problème vient évidemment du fait que les énoncés *compte:=* **succ** *compte* incorporés dans les tâches du type *incrémenteur* ne sont pas exécutés en exclusion mutuelle. Il conviendra donc d'examiner quels outils utiliser pour garantir l'exclusion mutuelle.

L'autre problème important est celui de la synchronisation des tâches; il peut arriver qu'une tâche ne puisse continuer son élaboration avant qu'une autre n'ait atteint un point donné de la sienne. On dit alors qu'il faut synchroniser ces deux tâches. Dans le programme *accès_mutuels*, le cas s'est d'ailleurs produit lorsque le programme principal devait attendre la fin de l'élaboration des tâches *incrémenteur* avant de pouvoir imprimer la valeur finale de la variable

compte. La méthode choisie dans ce cas pour assurer la synchronisation est évidemment primitive et coûteuse; là aussi, des outils plus performants devront être développés.

Le tableau suivant, qui sera complété par la suite, indique les notions analogues dans le traitement des coroutines et des paroutines.

coroutines	paroutines
process	**processor**
none	**nothing**
current	**curr_task**
exchange	**switch**
null	*inexistant*
attached	*ready, running*
detached	*waiting*
terminated	*finished, failed*

Au niveau du matériel, une instruction de base souvent réalisée pour construire des outils pour assurer l'exclusion mutuelle ou la synchronisation de tâches est le "Test and set". Cette instruction va placer une information dans une cellule de la mémoire tout en faisant un test sur son ancien contenu; il est garanti, au moyen d'un dispositif physique ad-hoc, que la tâche qui exécute cette instruction ne peut être interrompue entre le moment où l'on extrait l'ancien contenu de la cellule pour en tester la valeur et celui où l'on stocke sa nouvelle valeur. L'instruction "Test and set" est donc toujours exécutée en exclusion mutuelle; on dira qu'elle est indivisible.

En Newton, l'assignation inverse =*:* joue le rôle du "Test and set"; plus spécifiquement, il est garanti qu'il y a indivisibilité entre le moment où l'on extrait l'ancien contenu de la variable à droite du symbole =*:* et celui où l'on stocke son nouveau contenu. Il va de plus de soi que les instructions et clauses de manipulation de tâches, par exemple **switch**, sont indivisibles (de même d'ailleurs que celles de manipulation de coroutines). On verra, au moyen d'exemples, comment il est possible d'utiliser ces outils de base pour définir des outils de synchronisation et d'exclusion, mutuelle de plus haut niveau et d'un emploi plus commode.

Au moyen d'une utilisation judicieuse de l'assignation inverse, on va montrer une manière d'assumer l'exclusion mutuelle. On va réaliser pour celà une classe *verrou*; étant donné un objet *mut_excl* de cette classe, si deux séquences d'énoncés *s1* et *s2* sont encachées des primitives *mut_excl.fermer* et *mut_excl.ouvrir* il sera garanti que les exécutions de *s1* et *s2* seront séparées dans le temps (pour autant que *s1* et *s2* n'utilisent pas le verrou). La classe *verrou* répondra au protocole suivant :

```
class verrou
   value moi
   attribute libre, fermer, ouvrir
(* Un objet du type verrou est utilisé pour forcer l'exclusion mutuelle à une section
   critique. Les processeurs en attente de pouvoir exécuter leur section critique
   seront libérés dans un ordre non défini à priori.
*)
declare (*verrou*)
   Boolean variable ouvert value libre := true;
   (*false ssi un processeur est dans sa section critique *)
```

verrou **expression** *fermer* =
(* à exécuter avant d'entrer en section critique protégée par ce verrou; le résultat est le verrou concerné.
*)
/* ... */

verrou **expression** *ouvrir* =
(* à exécuter avant de quitter la section critique protégée par ce verrou; le résultat est le verrou concerné.
*)
/* ... */
do (**verrou**) **done**

On pourrait penser commencer la réalisation de la primitive *fermer* par une clause du genre

```
if ouvert then
   ouvert := false;
   /* ... */
default   /* ... */  done
```

Ceci est incorrect. Si deux tâches exécutent en même temps cette clause conditionnelle, toutes deux pourraient trouver que *ouvert* est vraie (il n'y a pas indivisibilité entre le test **if** *ouvert* et l'assignation *ouvert* := **false**). Ceci permettrait aux deux tâches d'entrer simultanément dans leur section critique. On peut, par contre, considérer la clause :

```
if false =: ouvert then
   /* ... */
default  /* ... */  done
```

La séquence d'énoncés entre **then** et **default** ne sera exécutée que si *ouvert* était initialement vraie; dans l'autre cas, il sera élaboré la séquence entre **default** et **done**. Vu l'indivisibilité énoncée de l'assignation inverse, si deux tâches cherchent à exécuter cette clause lorsque *ouvert* est vraie, seule l'une d'entre elles pourra exécuter la séquence entre **then** et **default**; l'autre exécutera celle entre **default** et **done**. Laquelle des deux tâches exécutera la séquence entre **then** et **default** qui lui permettra d'entrer dans sa section critique est d'ailleurs imprévisible. Entre **default** et **done** on pourrait alors faire mettre **curr_task** dans une file d'attente avant de l'interrompre; dans ce cas, l'opération *ouvrir* devrait (le cas échéant) libérer un élément de cette file d'attente. Dans un premier temps, on va proposer une implantation plus simple basée sur une attente dynamique. On considère la boucle suivante :

```
until false =: ouvert repetition
```

Une tâche qui aborde cette boucle lorsque *ouvert* est fausse va rester dans cette boucle jusqu'à ce qu'une autre tâche rende *ouvert* vraie. De plus, si deux tâches abordent cette boucle lorsque *ouvert* est vraie, seule l'une d'entre elles pourra la quitter immédiatement; l'autre devra attendre que *ouvert* redevienne vraie. Sous cette forme, l'attente dynamique sera très coûteuse : une tâche qui aborde le verrou avec *ouvert* fausse risque de gaspiller la majeure partie d'une tranche de temps dans cette boucle avant que le contrôle ne puisse être rendu à une autre tâche susceptible de rendre *ouvert* vraie. Pour autant que la section critique à protéger soit courte, le prix peut être rendu supportable en reformulant cette boucle comme suit :

```
until false =: ouvert repeat
   switch
repetition
```

Pour l'implantation de la primitive *ouvrir*, il suffira alors de réaliser l'assignation *ouvert*:=**true**.

On va examiner quelques pièges liés à l'utilisation de verrous. On suppose des déclarations de la forme suivante :

```
verrou value v = verrou;
paroutine tâche_1 do
    /* ... */
    v.fermer;
    /* ... section critique ... */
    v. ouvrir;
    /* ... */
done (* tache_1 *)

paroutine tâche_2 do
    /* ... */
    v.fermer;
    /* ... section critique ...*/
    v.fermer;
    /* ... */
done (* tâche_2 *)
```

On note les déclarations de paroutines *tâche_1* et *tâche_2*. Une déclaration de paroutine est l'analogue d'une déclaration de coroutine; elle dénote une tâche seule de son type qui est créée et connectée au point de sa déclaration. A partir d'une déclaration de paroutine, il y aura exécution parallèle de cette paroutine et du bloc contenant sa déclaration. Dans l'exemple précédent, *tâche_1* et *tâche_2* seront exécutées en parallèles. Par erreur, il a été programmé *v.fermer* au lieu de *v.ouvrir* à la fin de la section critique de *tâche_2*. Si *tâche_2* vient à exécuter sa section critique avant *tâche_1*, elle pourra exécuter cette dernière; par contre, *tâche_2* restera bloquée à l'énoncé *v.fermer* à la fin de sa section critique, aucune tâche n'étant à même d'ouvrir le verrou. La tâche *tâche_1*, quant à elle, restera bloquée à l'énoncé *v.fermer* au début de sa section critique. On dit qu'il y a un phénomène de blocage. Pour l'éviter, chaque clause *v.fermer* doit être suivi d'une clause *v.ouvrir* exécutée par la même tâche. Cette condition ne suffit pas; on considère, par exemple, le cas suivant :

```
verrou value v1 = verrou, v2 = verrou;

paroutine tâche_1 do
    /* ... */
    v1.fermer;
        /* ...protégé par v1 ... */
        v2.fermer;
            /* ... protégé par v1, v2 ... */
        v2. ouvrir;
        /* ... protégé par v1 ... */
    v1.ouvrir;
    /* ... */
done (* tâche_1 *);

paroutine tâche_2 do
    /* ... */
    v2.fermer;
        /* ... protégé par v2 ... */
        v1.fermer;
            /* ... protégé par v1, v2 ... */
        v1.ouvrir;
        /* ... protégé par v2 ... */
    v2.ouvrir;
    /* ... */
done (* tâche_2 *)
```

Si les paroutines *tâche_1* et *tâche_2* abordent en même temps les sections critiques protégées par *v1.fermer* et respectivement *v2.fermer*, elles entreront dans ces sections critiques. Par

contre *tâche_1* ne pourra avancer au-delà de son *v2.fermer* et *tâche_2* au-delà de son *v1.fermer*; les deux tâches se bloquent mutuellement. On dit qu'il y a un phénomène d'interblocage.

Un problème classique, à traiter en programmation parallèle, est celui des philosophes affamés. Quelques philosophes sont assis autour d'une table circulaire; chacun d'entre eux dispose d'une assiette. Entre chaque paire d'assiettes, donc de philosophes, il a été placé une fourchette. Au milieu de la table figure un grand bol de spaghettis (ce bol est supposé inépuisable; fig. 63).

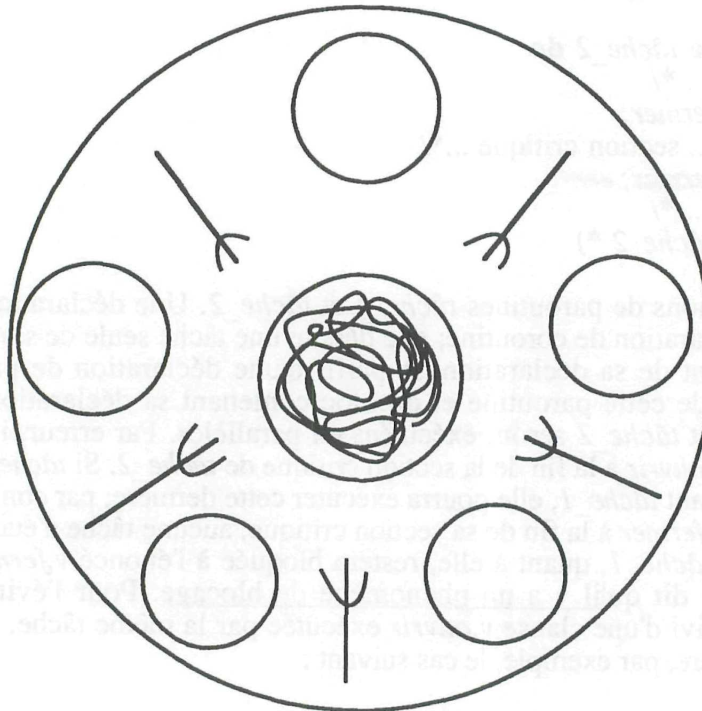


Fig. 63

Alternativement, pendant un intervalle de temps plus ou moins long, un philosophe pense et mange. Pour pouvoir manger, il doit être à même d'utiliser deux fourchettes; il ne peut donc manger que si les fourchettes placées à sa gauche et à sa droite sont libres (donc, si aucun des ses voisins immédiat ne mange). On suppose qu'une fois qu'il a fini de manger, un philosophe replace correctement à sa gauche et à sa droite les deux fourchettes qu'il a utilisées. Pour le reste, les philosophes ne communiquent pas entre eux.

Pour programmer cette application, il est nécessaire d'éviter certains écueils. On va tout d'abord invoquer quelques solutions incorrectes.

- Lorsqu'il a faim, un philosophe accapare, dès qu'elle devient libre, l'une des deux fourchettes, puis l'autre.

  Ceci ne va pas si tous les philosophes ont faim au même instant et qu'ils commencent tous par saisir la fourchette à leur gauche. Il y a alors interblocage général.

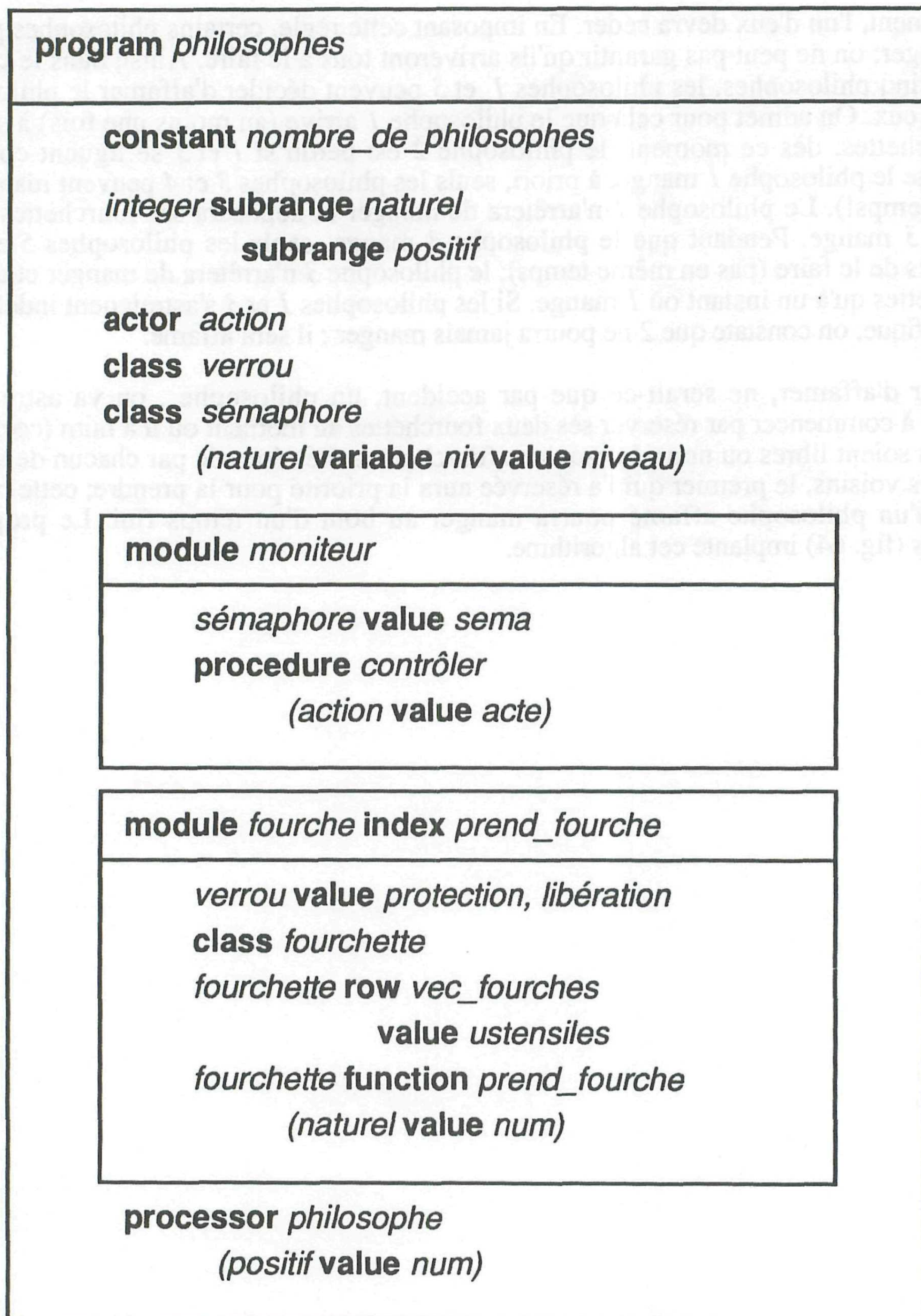
- On pourrait imposer au philosophe affamé de redéposer la fourchette qu'il a saisie s'il n'arrive pas, dans un délai donné, à prendre l'autre.

  Ceci n'est pas satisfaisant non plus dans le sens qu'avec un peu de malchance, tous les philosophes pourraient alternativement se saisir de leur fourchette gauche et la redéposer aux mêmes moments.

Pour éviter ceci, on pourrait obliger les philosophes à prendre leurs deux fourchettes au même instant. Evidemment, si deux philosophes voisins cherchent à prendre la même fourchette au

même moment, l'un d'eux devra céder. En imposant cette règle, certains philosophes pourront certes manger; on ne peut pas garantir qu'ils arriveront tous à le faire. Ainsi, dans le cas d'une tablée de cinq philosophes, les philosophes *1* et *3* peuvent décider d'affamer le philosophe 2 assis entre eux. On admet pour celà que le philosophe *1* arrive (au moins une fois) à saisir ses deux fourchettes; dès ce moment, le philosophe 2 est perdu si *1* et *3* se liguent contre lui. Pendant que le philosophe *1* mange, à priori, seuls les philosophes *3* et *4* peuvent manger (pas en même temps!). Le philosophe *1* n'arrêtera de manger et déposera ses fourchettes qu'à un instant où *3* mange. Pendant que le philosophe *3* mange, seuls les philosophes *5* et *1* sont susceptibles de le faire (pas en même temps); le philosophe *3* n'arrêtera de manger et déposera ses fourchettes qu'à un instant où *1* mange. Si les philosophes *1* et *3* s'astreignent indéfiniment à cette politique, on constate que *2* ne pourra jamais manger : il sera affamé.

Pour éviter d'affamer, ne serait-ce que par accident, un philosophe , on va astreindre le philosophe à commencer par réserver ses deux fourchettes au moment où il a faim (ceci que les fourchettes soient libres ou non). Lorsqu'une fourchette a été réservée par chacun de ses deux philosophes voisins, le premier qui l'a réservée aura la priorité pour la prendre; cette politique garantit qu'un philosophe affamé pourra manger au bout d'un temps fini. Le programme *philosophes* (fig. 64) implante cet algorithme.



```
program philosophes

    constant nombre_de_philosophes
    integer subrange naturel
            subrange positif

    actor action
    class verrou
    class sémaphore
        (naturel variable niv value niveau)

    module moniteur

        sémaphore value sema
        procedure contrôler
            (action value acte)

    module fourche index prend_fourche

        verrou value protection, libération
        class fourchette
        fourchette row vec_fourches
                value ustensiles
        fourchette function prend_fourche
            (naturel value num)

    processor philosophe
        (positif value num)
```

Fig. 64

Dans ce programme, on commence par définir quelques outils de synchronisation et d'exclusion mutuelle. On reconnaît tout d'abord la classe *verrou* dont la conception a été discutée au préalable.

```
philosophes                                        Vax Newton Compiler 1.0
                                           Page     1
                Source listing


   1
   1   /* /*OLDSOURCE=USER2:[RAPIN]PHILOSOPHES.NEW*/ */
   1   PROGRAM philosophes DECLARE
   4     CONSTANT nombre_de_philosophes=5;
   9
   9     integer SUBRANGE naturel(naturel>=0)
  17             SUBRANGE positif(positif>0);
  25
  25     ACTOR action;
  28
  28     CLASS verrou  VALUE moi
  32       ATTRIBUTE libre,fermer,ouvrir
  38     (*Un objet du type  verrou  est utilise pour forcer l'exclusion
  38       mutuelle a une section critique. Les processeurs en attente
  38       de pouvoir executer leur section critique seront liberes dans
  38       un ordre non defini a priori
  38     *)
  38     DECLARE(*verrou*)
  39       Boolean VARIABLE ouvert VALUE libre:=TRUE;
  47       (*FALSE ssi un processeur est dans sa section critique*)
  47
  47       verrou EXPRESSION fermer=
  51       (*A executer avant d'entrer en section critique protegee par
  51         le verrou concerne; le resultat est le verrou concerne
  51       *)
  51         (UNTIL FALSE=:ouvert REPEAT SWITCH REPETITION; moi);
  63
  63       verrou EXPRESSION ouvrir=
  67       (*A executer avant de quitter la section critique protegee par
  67         le verrou concerne; le resultat est le verrou concerne
  67       *)
  67         (ouvert:=TRUE; moi)
  74     DO(*verrou*)DONE;
  77     /* /*EJECT*/ */
```

La classe *sémaphore* définit un outil de synchronisation et d'exclusion mutuelle un peu plus sophistiqué. La notion de sémaphore a été introduite par Dijkstra; dans le langage Algol-68, les tâches concurrentes sont gérées au moyen de sémaphores. Un sémaphore *séma* = *sémaphore(n)* permet de réaliser des sections critiques élaborables en parallèles par *n* tâches, mais pas plus. Avant d'entrer dans sa section critique, une tâche effectuera l'énoncé *séma.baisser*; au moment de le quitter, elle effectuera *séma.lever*.

```
philosophes                                    Vax Newton Compiler 1.0
                                    Page     2
                 Source listing

   CLASS semaphore  VALUE moi
     ATTRIBUTE niveau,occupe,baisser,lever
      (naturel VARIABLE niv VALUE niveau)
  (*Un objet  semaphore(niv)  sera utilise pour proteger une section
    critique a laquelle au maximum  niv  processeurs peuvent acceder
    simultanement. La valeur courante  niveau  indique le nombre de
    processeurs autorises a entrer en section critique. Les proces-
    seurs qui attendent de pouvoir executer leur section critique
    seront liberes dans l'ordre premier venu, premier servi
   *)
   DECLARE(*semaphore*)
     verrou VALUE mut_excl=verrou;

     OBJECT liste(task VALUE processeur; liste VARIABLE suiv)
       VARIABLE tete:=NIL;
     liste REFERENCE ins->tete;

     Boolean EXPRESSION occupe=
      (*TRUE ssi des processeurs attendent de pouvoir entrer dans
        leur section critique
      *)
        tete~=NIL;

     semaphore EXPRESSION baisser=
      (*A executer avant d'entrer dans une section critique protegee
        par ce semaphore; le resultat est le semaphore concerne  moi
      *)
        (mut_excl.fermer;
         IF niveau>0 THEN
           niv:=PRED niv
         DEFAULT
           ins->(ins:=liste(CURR_TASK,NIL)).suiv;
           mut_excl.ouvrir
         INTERRUPT DONE;   ?=> l.206
         mut_excl.ouvrir;
        moi);

     semaphore EXPRESSION lever=
      (*A executer avant de quitter une section critique protegee
        par ce semaphore; le resultat est le semaphore concerne  moi
      *)
        (mut_excl.fermer;
         CONNECT tete THEN
           IF (tete:=suiv)=NIL THEN ins->tete DONE;
           UNTIL STATE processeur=waiting REPEAT SWITCH REPETITION;
           SCHEDULE processeur NOW
         DEFAULT
           niv:=SUCC niv;
           mut_excl.ouvrir
         DONE;
        moi)
   DO(*semaphore*)DONE;
```

```
  234    /* /*EJECT*/ */
```

Dans l'implantation de la classe *sémaphore*, on remarque des nouvelles primitives. Tout d'abord le type *task*, analogue de *activity*; toute valeur d'un type processeur (donc toute tâche) peut être convertie dans le type *task*. Comme d'habitude, la conversion inverse d'une valeur du type *task* en une valeur d'un type processeur donné nécessite un contrôle à l'exécution. La clause **interrupt**, analogue à **return**, a pour effet d'arrêter l'élaboration de la tâche courante **curr_task** qui passe à l'état *waiting*; à ce stade, une des tâches dans l'état *ready* passe à l'état *running* (le processeur physique attaché à la tâche interrompue lui est attribué). Si, au moment d'élaborer une clause **interrupt**, il ne reste aucune tâche à l'état *ready* ou *running*, l'exécution de l'ensemble du programme est terminée. L'énoncé **schedule**, analogue de **activate**, permet de réactiver une tâche en attente; cette dernière passe de l'état *waiting* à l'état *ready* ou *running*; elle poursuit donc son élaboration parallèlement à celle qui a effectué l'énoncé **schedule**.

On constate qu'un sémaphore fait usage d'une file d'attente dans laquelle les tâches en attente sont interrompues et de laquelle elles sont débloquées dans l'ordre premier venu, premier servi. Les opérations sur cette file d'attente sont rendues indivisibles au moyen du verrou *mut_excl*. Ce système ne présente d'attente dynamique que pendant qu'une tâche attend que ce verrou soit libre avant d'effectuer une section critique. Du fait qu'une tâche sera passive pendant la plus grande partie de son attente, il n'y a pas d'inconvénient d'utiliser un sémaphore pour assurer l'exclusion mutuelle sur une section critique longue. Le lecteur examinera attentivement les points où le verrou *mut_excl* est fermé et ouvert. Il est fermé au début du code des primitives *baisser* et *lever*; cette opération est suivie d'un énoncé conditionnel : il faut s'assurer que le verrou sera ouvert dans chaque schéma d'exécution. Bien entendu, l'énoncé *mut_excl.ouvrir* avant la clause **interrupt** est essentielle : d'autres tâches doivent pouvoir accéder au sémaphore, ne serait-ce que pour débloquer la tâche interrompue. Dans la fonction *lever*, on remarque que si *tête* n'est pas initialement vide, il n'y a aucun énoncé *mut_excl.ouvrir* entre le **connect** *tête* et le **default** correspondant : le lecteur vérifiera que c'est la tâche débloquée au moyen de l'énoncé **schedule** *processeur* **now** qui se chargera d'ouvrir le verrou en exécutant la partie finale de l'opération *baisser*.

Il est possible de créer un objet de la forme *sémaphore (0)*; ceci peut être utile pour synchroniser des tâches. Soit *synchr* un sémaphore initialisé de cette manière-là : une tâche qui doit attendre qu'une autre ait atteint un certain point de son élaboration avant de pouvoir continuer exécutera l'énoncé *synchr.baisser*; l'autre tâche effectuera l'opération *synchr.lever* lorsqu'elle atteint le point qui permet de débloquer la première tâche :

```
semaphore value synchr = sémaphore (0);
integer variable var;

paroutine tâche_1 do
    /* ... */
    read (var);
    synchr.lever;
    /* ... */
done (* tâche_1 *);

paroutine tâche_2 do
    /* ... */
    synchr.baisser;
    print (var);
    /* ... */
done (* tâche_2 *)
```

Dans cet exemple, la paroutine *tâche_2* ne pourra dépasser l'énoncé *synchr.baisser* tant que *tâche_1* n'a pas atteint l'instruction *synchr.lever* et par conséquent qu'elle ait lu une valeur dans la variable *var*.

```
philosophes                                    Vax Newton Compiler 1.0
                                          Page     4
                 Source listing

 MODULE moniteur
    ATTRIBUTE controler
 DECLARE(*moniteur*)
    semaphore VALUE sema=semaphore(1);

    PROCEDURE controler
       (action VALUE acte)
    (*effectue en exclusion mutuelle l'action donnee  acte ;
       le cas echeant, les processeurs executeront leur section
       critique dans l'ordre premier venu, premier servi
    *)
    DO sema.baisser; acte EVAL; sema.lever DONE(*controler*)
 DO(*moniteur*)DONE;
 /* /*EJECT*/ */
```

Le module *moniteur* exporte la procédure *contrôler*; l'exécution de l'objet procédural *acte* sera effectuée en exclusion mutuelle garantie par le sémaphore *sema*. Certains langages de programmation, en particulier Portal, utilisent le moniteur comme outil de synchronisation et d'exclusion mutuelle; un moniteur est un module dont toutes les procédures et fonctions exportées sont exécutées en exclusion mutuelle. Un moniteur peut être implanté en y incorporant un sémaphore qui sera baissé du début de l'exécution de chaque procédure exportée et levée à la fin de cette dernière. Des précautions sont à prendre si une telle procédure ou fonction exportée est de nature récursive : si l'on rebaisse le sémaphore au début d'une invocation récursive, il y aura évidemment blocage. On pourra l'éviter par l'introduction d'une procédure auxiliaire :

```
module moniteur
      attribute proc
declare
      sémaphore value séma = sémaphore (1);

      procedure corps_proc
            (integer value n)
      do (* corps_proc *)
            /* ... */
            if n > 0 then
                  /* ... */
                  corps_proc (n-1)
            done;
            /* ... */
      done (* corps_proc *);

      procedure proc (integer value n) do
            sema.baisser; corps_proc (n); sema.lever
      done (* proc*);

      /* ... */
do  /* ... */  done (* moniteur *)
```

Dans cet exemple, le module *moniteur* exporte la procédure *proc* dont l'exécution est réalisée en exclusion mutuelle sous la protection du sémaphore *séma*; l'algorithme récursif correspondant a été entièrement programmé sous la forme de la procédure *corps_proc* non exportée et dont la partie exécutable ne touche pas au sémaphore : les appels récursifs de cette procédure n'entraîneront donc aucun blocage.

```
philosophes                                        Vax Newton Compiler 1.0
                                              Page    5
                   Source listing

MODULE fourche
  INDEX prend_fourche  ATTRIBUTE fourchette
(*definit une ensemble de  nombre_de_philosophes  fourchettes*)
DECLARE(*fourchette*)
  verrou VALUE protection=verrou,liberation=verrou;

  CLASS fourchette
    ATTRIBUTE reserver,utiliser,liberer
  DECLARE(*fourchette*)
    task QUEUE tq VALUE clients=tq(2);
 (*pour les philosophes qui attendent d'utiliser cette fourchette*)

    Boolean VARIABLE libre:=TRUE;

    PROCEDURE reserver DO
    (*avant de se saisir d'une fourchette, un philosophe doit la
      reserver
    *)
      protection.fermer;
      clients APPEND CURR_TASK;
      protection.ouvrir
    DONE(*reserver*);

    PROCEDURE utiliser DECLARE
    (*le philosophe CURR_TASK saisit la fourchette concernee*)
      task VARIABLE moi
    DO(*utiliser*)
      protection.fermer;
      IF TAKE
        IF clients FRONT=CURR_TASK THEN
          ~libre
        DEFAULT TRUE DONE
      THEN
        protection.ouvrir INTERRUPT
      DEFAULT
        moi FROM clients; libre:=FALSE;
        protection.ouvrir
      DONE
    DONE(*utiliser*);

    PROCEDURE liberer DECLARE
    (*le philosophe CURR_TASK libere la fourchette concernee*)
      task VARIABLE suiv
    DO(*liberer*)
      protection.fermer;
      UNLESS EMPTY clients THEN
        suiv FROM clients;
        protection.ouvrir;
        liberation.fermer;
        UNTIL STATE suiv=waiting REPEAT SWITCH REPETITION;
        SCHEDULE suiv NOW;
        liberation.ouvrir
```

```
philosophes                                    Vax Newton Compiler 1.0
                                          Page    6
                 Source listing

       DEFAULT
         libre:=TRUE;
         protection.ouvrir
       DONE
     DONE(*liberer*)
  DO(*fourchette*)DONE ROW vec_fourches VALUE
     ustensiles=
       THROUGH
         vec_fourches(0 TO PRED nombre_de_philosophes):=fourchette
       REPETITION;

  fourchette FUNCTION prend_fourche
     (naturel VALUE num)
  (*la fourchette de numero donne*)
  DO TAKE ustensiles[num] DONE
DO(*fourche*)DONE;
/* /*EJECT*/ */
```

Dans la module *fourche*, il est défini la classe *fourchette* et la rangée de fourchettes *ustensiles*. La plupart des opérations sur les fourchettes sont protégées par le verrou *protection* (on remarque que les sections critiques sont courtes). Une fourchette comporte la queue *clients* dans laquelle sont insérés les philosophes qui ont réservé la fourchette avant de pouvoir l'utiliser; une fourchette donnée ne pouvant être convoitée que par deux philosophes, une queue de dimension 2 suffit.

Dans la procédure libérer, on remarque que le déblocage du philosophe *suiv* est réalisé dans une section critique protégée par le verrou *libération*. Ceci vient du fait que les deux philosophes qui encadrent un philosophe affamé peuvent libérer leurs fourchettes au même moment et tenter de le débloquer au même instant. En l'absence de ce verrouillage, l'un d'eux pourrait exécuter l'énoncé **schedule** *suiv* **now** lorsque *suiv* n'est plus dans l'état *waiting*, entraînant par là une erreur d'exécution (à noter que ce cas est rare et insidieux!).

```
philosophes                                          Vax Newton Compiler 1.0
                                             Page     7
                     Source listing

     PROCESSOR philosophe(positif VALUE num)DECLARE
       CONSTANT temps_reflexion=6_500,temps_repas=3_500;
       (*un philosophe ne fait que penser et manger; il est
         indique le temps que ca lui prend en moyenne pour
         chacune de ces activites
       *)

       PROCEDURE reflechir DO
         FOR integer FROM 1 BY 1 TO temps_reflexion*poisson REPETITION
       DONE(*reflechir*);

       fourchette VALUE f1=fourche[(PRED num)\nombre_de_philosophes],
                        f2=fourche[num\nombre_de_philosophes];
       (*lorsqu'il mange, le philosophe doit necessairement utiliser
         ces deux fourchettes-la
       *)

       PROCEDURE manger DO
         controler
           (BODY action DO
              print("Le philosophe"_,edit(num,2,0),_"a faim",line)
            DONE);
         f1.reserver; f2.reserver;
         (*avant de pouvoir les utiliser, le philosophe doit reserver
           ses deux fourchettes
         *)
         f1.utiliser; f2.utiliser;
         (*le philosophe a pris possession de ses fourchettes; il
           peut manger
         *)
         controler
           (BODY action DO
            print("Le philosophe"_,edit(num,2,0),_"commence a manger",
                    line)
            DONE);
         FOR integer FROM 1 BY 1 TO temps_repas*poisson REPETITION;
         (*le philosophe a termine son repas*)
         controler
           (BODY action DO
             print("Le philosophe"_,edit(num,2,0),_"a fini de manger",
                    line)
            DONE);
         f1.liberer; f2.liberer
        (*le philosophe ne doit pas oublier de rendre ses fourchettes*)
       DONE(*manger*)
     DO(*philosophe*)LOOP
       reflechir; manger
     REPETITION(*philosophe*)DONE
     /* /*EJECT*/ */
```

Dans l'algorithme des philosophes, incorporé dans le type processeur *philosophe*, on remarque que les impressions sont exécutées en exclusion mutuelle assurée par la procédure *contrôler* du module *moniteur*. Un énoncé *print* comportant plusieurs clauses d'impression n'est en effet pas indivisible. Une impression étant une opération lente, il est utile que le moniteur utilise un sémaphore et non un verrou pour assurer l'exclusion mutuelle, d'autant que l'on veut que les impressions aient lieu dans l'ordre de leur arrivée dans le moniteur.

```
philosophes                                   Vax Newton Compiler 1.0
                                        Page     8
                  Source listing

   DO(*philosophes*)
     FOR integer VALUE n FROM 1 TO nombre_de_philosophes REPEAT
       philosophe(n)
     REPETITION
   DONE(*philosophes*)

   ****  No messages were issued  ****
```

La partie exécutable du programme *philosophes* ne fait que créer les philosophes avant de s'éteindre. Ensuite tout se passe entre les philosophes qui vont exécuter, en parallèle, leur algorithme indéfiniment.

Il est donné un début d'exécution de ce programme; on peut vérifier qu'il est conforme aux règles et qu'un philosophe affamé arrive toujours (tôt ou tard) à manger. Même en l'absence d'un *randomize*, différentes exécutions du programme donnent lieu à des résultats différents (bien que du même style) du fait de l'indéterminisme lié au parallélisme.

```
Le philosophe  3 a faim
Le philosophe  1 a faim
Le philosophe  1 commence a manger
Le philosophe  2 a faim
Le philosophe  4 a faim
Le philosophe  4 commence a manger
Le philosophe  5 a faim
Le philosophe  4 a fini de manger
Le philosophe  1 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  5 a fini de manger
Le philosophe  2 commence a manger
Le philosophe  5 a faim
Le philosophe  5 commence a manger
Le philosophe  4 a faim
Le philosophe  2 a fini de manger
Le philosophe  3 commence a manger
Le philosophe  1 a faim
Le philosophe  2 a faim
Le philosophe  3 a fini de manger
Le philosophe  5 a fini de manger
Le philosophe  4 commence a manger
Le philosophe  1 commence a manger
Le philosophe  1 a fini de manger
Le philosophe  2 commence a manger
Le philosophe  2 a fini de manger
Le philosophe  5 a faim
Le philosophe  3 a faim
Le philosophe  4 a fini de manger
Le philosophe  2 a faim
Le philosophe  3 commence a manger
Le philosophe  5 commence a manger
Le philosophe  5 a fini de manger
Le philosophe  1 a faim
Le philosophe  3 a fini de manger
Le philosophe  5 a faim
Le philosophe  2 commence a manger
Le philosophe  2 a fini de manger
Le philosophe  1 commence a manger
Le philosophe  2 a faim
Le philosophe  1 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  2 commence a manger
Le philosophe  4 a faim
Le philosophe  2 a fini de manger
Le philosophe  5 a fini de manger
Le philosophe  4 commence a manger
Le philosophe  4 a fini de manger
Le philosophe  4 a faim
Le philosophe  4 commence a manger
Le philosophe  3 a faim
Le philosophe  4 a fini de manger
Le philosophe  3 commence a manger
Le philosophe  2 a faim
Le philosophe  5 a faim
Le philosophe  5 commence a manger
Le philosophe  5 a fini de manger
Le philosophe  5 a faim
Le philosophe  1 a faim
Le philosophe  4 a faim
Le philosophe  3 a fini de manger
Le philosophe  2 commence a manger
Le philosophe  2 a fini de manger
Le philosophe  1 commence a manger
Le philosophe  1 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  5 a fini de manger
Le philosophe  4 commence a manger
Le philosophe  3 a faim
Le philosophe  5 a faim
Le philosophe  4 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  3 commence a manger
Le philosophe  3 a fini de manger
Le philosophe  5 a fini de manger
Le philosophe  3 a faim
Le philosophe  1 a faim
Le philosophe  1 commence a manger
Le philosophe  5 a faim
Le philosophe  3 commence a manger
Le philosophe  3 a fini de manger
Le philosophe  1 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  2 a faim
Le philosophe  3 a faim
Le philosophe  2 commence a manger
Le philosophe  5 a fini de manger
Le philosophe  2 a fini de manger
Le philosophe  3 commence a manger
Le philosophe  5 a faim
Le philosophe  5 commence a manger
Le philosophe  4 a faim
Le philosophe  1 a faim
Le philosophe  2 a faim
Le philosophe  3 a fini de manger
Le philosophe  5 a fini de manger
Le philosophe  4 commence a manger
Le philosophe  3 a faim
Le philosophe  1 commence a manger
Le philosophe  4 a fini de manger
Le philosophe  5 a faim
Le philosophe  1 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  4 a faim
Le philosophe  2 commence a manger
Le philosophe  2 a fini de manger
Le philosophe  1 a faim
Le philosophe  3 commence a manger
Le philosophe  5 a fini de manger
Le philosophe  1 commence a manger
Le philosophe  2 a faim
Le philosophe  3 a fini de manger
Le philosophe  4 commence a manger
Le philosophe  4 a fini de manger
Le philosophe  5 a faim
Le philosophe  4 a faim
Le philosophe  1 a fini de manger
Le philosophe  5 commence a manger
Le philosophe  3 a faim
Le philosophe  2 commence a manger
```

Il est possible de formuler une version parallèle de l'algorithme de tri "quick-sort" dû à Hoare et présenté au chapitre 5. On rappelle que cet algorithme prend un élément de la rangée à trier; au moyen d'un parcours en zig-zag de la rangée, il place cet élément à son endroit définitif en le faisant précéder des éléments qui lui sont inférieurs et suivre de ceux qui lui sont supérieurs. Il suffit alors de trier récursivement les deux sous-rangées au moyen du même algorithme. Rien n'empêche d'élaborer en parallèle ces deux tris récursifs: ils sont indépendants l'un de l'autre. Il faudra simplement s'assurer que ces deux tris soient achevés avant de rendre le contrôle au programme appeleur.

On a donc là un problème de synchronisation typique : lancer un certain nombre de tâches et suspendre sa propre exécution jusqu'à ce que chacune de ces tâches soient terminée. Dans le programme *tris_taches* suivant, il a été utilisé des objets de la classe *salle_d_attente* pour assurer cette synchronisation.

Dans l'algorithme *trier*, la procédure *tri_sec* est remplacée par une déclaration de processeur. Trois paramètres sont communiqués aux tâches correspondantes au moment de leur création : les deux indices spécifiant la sous-rangée que la tâche est chargée de trier ainsi qu'un objet *père* du type *salle_d_attente*.

```
tris_taches                                  Vax Newton Compiler 1.0
                                Page    1
             Source listing


    1   /* /*OLDSOURCE=USER2:[RAPIN]TRIS_TACHES.NEW*/ */
    1   PROGRAM tris_taches DECLARE
    4     integer SUBRANGE naturel(naturel>=0);
   13
   13     CLASS verrou   VALUE moi
   17       ATTRIBUTE libre,fermer,ouvrir
   23     (*Un objet du type verrou  est utilise pour forcer l'exclusion
   23       mutuelle a une section critique. Les processeurs en attente
   23       de pouvoir executer leur section critique seront liberes dans
   23       un ordre non defini a priori
   23     *)
   23     DECLARE(*verrou*)
   24       Boolean VARIABLE ouvert VALUE libre:=TRUE;
   32       (*FALSE ssi un processeur est dans sa section critique*)
   32
   32       verrou EXPRESSION fermer=
   36       (*A executer avant d'entrer en section critique protegee par
   36         le verrou concerne; le resultat est le verrou concerne
   36       *)
   36         (UNTIL FALSE=:ouvert REPEAT SWITCH REPETITION; moi);
   48
   48       verrou EXPRESSION ouvrir=
   52       (*A executer avant de quitter la section critique protegee par
   52         le verrou concerne; le resultat est le verrou concerne
   52       *)
   52         (ouvert:=TRUE; moi)
   59     DO(*verrou*)DONE;
   62     /* /*EJECT*/ */
```

```
CLASS salle_d_attente  VALUE moi
  ATTRIBUTE controleur,dependants,en_attente,incr,decr,attendre
DECLARE(*salle_attente*)
  task VALUE controleur=CURR_TASK;

  naturel VARIABLE compte VALUE dependants:=0;
  (*le nombre de taches que le controleur devra attendre*)
  Boolean VARIABLE attente VALUE en_attente:=FALSE;
  (*vrai ssi le controleur est en attente*)
  verrou VALUE mut_excl=verrou;

  PROCEDURE erreur(string VALUE message)DO
    print(line,"###",message,"###",line)
  INTERRUPT(*erreur*)DONE;

  salle_d_attente FUNCTION incr DO
  (*incremente le nombre de taches dependantes; le resultat
    est la salle d'attente concernee

    Condition d'emploi: CURR_TASK=controleur
  *)
    IF CURR_TASK=controleur THEN
      CONNECT mut_excl.fermer THEN
        compte:=SUCC compte;
        ouvrir
      DONE
    DEFAULT
      erreur("'incr' execute par dependant")
    DONE
  TAKE moi DONE(*incr*);

  salle_d_attente FUNCTION decr DO
  (*diminue d'une unite le nombre de taches dependantes; relance
    le controleur si ce nombre devient nul. Le resultat est la
    salle d'attente concernee.

    Condition d'emploi: CURR_TASK~=controleur
  *)
    UNLESS CURR_TASK=controleur THEN
      CONNECT mut_excl.fermer THEN
        IF (compte:=PRED compte)=0 THEN
          IF en_attente THEN
          UNTIL STATE controleur=waiting REPEAT SWITCH REPETITION;
            attente:=FALSE;
            ouvrir;
            SCHEDULE controleur NOW
          DEFAULT ouvrir DONE
        DEFAULT ouvrir DONE
      DONE
    DEFAULT erreur("'decr' execute par controleur") DONE
  TAKE moi DONE;
```

```
tris_taches                                    Vax Newton Compiler 1.0
                                    Page     3
                 Source listing

   salle_d_attente FUNCTION attendre DO
    (*s'il y a des dependants, le controleur attend qu'ils aient
      tous descrementes le controleur; le resultat est la salle
      d'attente concernee.

      Condition d'emploi: CURR_TASK=controleur
    *)
      IF CURR_TASK=controleur THEN
         CONNECT mut_excl.fermer THEN
            IF dependants>0 THEN
               attente:=TRUE;
               ouvrir
            INTERRUPT DONE;
            ouvrir
         DONE
      DEFAULT
         erreur("'attendre' execute par dependant")
      DONE
    TAKE moi DONE(*attendre*)
 DO(*salle_d_attente*)DONE;
 /* /*EJECT*/ */
```

Chaque tâche *tri_sec* comporte une salle d'attente *fils*. Lorsque cette tâche crée les deux sous-tâches chargées de trier les deux sous-rangées résultant du parcours en zig-zag, elle associe à leur paramètre *père* sa salle d'attente *fils* en lui appliquant l'opération *incr* (il est important que cette opération soit réalisée par la tâche appelante). Après avoir lancé les sous-tâches, elle effectue l'opération *fils.attendre* qui la met en attente jusqu'à ce que les deux sous-tâches aient achevé leur élaboration. A la fin de son élaboration, une tâche *tri_sec* doit effectuer l'opération *père.decr* (il est important que cette opération soit réalisée par la tâche appelée).

Un objet de la classe *salle_d_attente* comporte les éléments suivants. On a tout d'abord la tâche *contrôleur* qui a créé la salle d'attente et qui est la seule autorisée à l'incrémenter. On a un compteur entier *compte* de valeur *dépendants* de valeur égale au nombre de sous-tâches que le contrôleur a créé et qui n'ont pas encore achevé leur exécution. La variable *attente* de valeur *en_attente* est vraie ssi la tâche *contrôleur* s'est suspendue après avoir exécuté la primitive *attendre*. Les sections critiques liées aux modifications d'état des variables *compte* et *attente* sont protégées par le verrou *mut_excl* (on vérifie que ces sections critiques sont courtes).

```
real ROW vecteur;

Boolean FUNCTOR relation(real VALUE gauche,droite)VALUE
  croissant=BODY relation DO TAKE gauche<droite DONE,
  decroissant=BODY relation DO TAKE gauche>droite DONE,
  absolu=BODY relation DO TAKE ABS gauche<ABS droite DONE,
  classes_entieres=
    BODY relation DO TAKE FLOOR gauche<FLOOR droite DONE;

PROCEDURE trier
   (vecteur VALUE tab; relation VALUE inf)
(*Rearrange les composantes du vecteur donne  tab  de maniere
  telle que l'on ait:

    LOW tab<=j/\j<k/\k<=HIGH tab IMPL ~inf[tab[K],tab[j]]

  Conditions d'emploi:

  inf  denote une relation d'ordre (eventuellement partielle)
       sur les valeur reelles; pour  x, y, z  reels, on doit
       avoir:

  inf[x,y] NAND inf[y,x] ;
  inf[x,y] NOR inf[y,x] IMPL (inf[x,z]==inf[y,z]) ;
  inf[x,y]/\inf[y,z] IMPL inf[x,z]
*)
DECLARE(*trier*)
  salle_d_attente VALUE separation_primaire=salle_d_attente;

  PROCESSOR tri_sec
    (salle_d_attente VALUE pere; integer VALUE bas,haut)
  (*Ce processeur est charge de trier la section  tab[bas] ,
    tab[haut] . La salle d'attente  pere  aura ete incrmentee
    par la tache appelante; avant de terminer son execution,
    ce processeur doit decrementer la salle d'attente
  *)
  DECLARE
    salle_d_attente VALUE fils=salle_d_attente;
    real VALUE elt=tab[bas];
    integer VARIABLE b:=bas, h:=SUCC haut
  DO(*tri_sec*)
    (*Invariants:
      bas<=k/\k<=b IMPL ~inf[elt,tab[k]]
      h<=k/\k<=haut IMPL ~inf[tab[k],elt]
    *)
    CYCLE zig_zag REPEAT
      WHILE
        inf[elt,tab[(h:=PRED h)]]
      REPETITION;
    IF b=h EXIT zig_zag DONE;
      tab[b]:=:tab[h];
      WHILE
```

```
tris_taches                                        Vax Newton Compiler 1.0
                                                   Page      5
                    Source listing

              inf[tab[(b:=SUCC b)],elt]
            REPETITION;
          IF b=h EXIT zig_zag DONE;
            tab[b]:=:tab[h]
          REPETITION;
          IF bas<(b:=PRED b) THEN tri_sec(fils.incr,bas,b) DONE;
          IF (h:=SUCC h)<haut THEN tri_sec(fils.incr,h,haut) DONE;
          fils.attendre; pere.decr
        DONE(*tri_sec*)
      DO(*trier*)
        tri_sec(separation_primaire.incr,LOW tab,HIGH tab);
        separation_primaire.attendre
      DONE(*trier*);

      PROCEDURE imprimer(vecteur VALUE vec) DO
        THROUGH vec INDEX k VALUE vk REPEAT
          IF k\5=1 THEN line DONE; edit(vk,12,8)
        REPETITION
      DONE(*imprimer*);

      PROCEDURE traiter
        (integer VALUE taille; real EXPRESSION terme;
         relation VALUE ordre)
      DECLARE(*traiter*)
        vecteur VALUE vec=
          THROUGH vecteur(1 TO taille):=terme REPETITION;
      DO(*traiter*)
        print(page,"***Vecteur original***");
        imprimer(vec);
        trier(vec,ordre);
        print(line,"***Vecteur trie***");
        imprimer(vec); print(line,"<<<FIN>>>")
      DONE(*traiter*)
    DO(*tris_taches*)
      randomize;
      traiter(100,random,croissant);
      traiter(100,random,decroissant);
      traiter(50,normal,absolu);
      traiter(50,10*poisson,classes_entieres)
    DONE(*tris_taches*)

    ****  No messages were issued ****
```

La programmation des primitives *incr*, *decr* et *attendre* est alors relativement claire. Pour *incr*, il suffit d'augmenter d'une unité la variable *compte*. Inversement, *decr* diminuera ce même compteur d'unité; s'il devient nul et que *en_attente* est vraie, la tâche *contrôleur* est relancée. Lorsque le contrôleur exécute la primitive *attendre*, il s'interrompt après avoir vérifié que *dépendants* est (encore) supérieur à zéro et après avoir rendu *en_attente* vraie.

Pour le reste, ce programme est identique au programme *tris* du chapitre 5. Ses résultats sont de la même forme. Un exemple d'exécution suit.

```
***Vecteur original***
    .99886494    .66371453    .89372808    .73250770    .78642405
    .35801113    .09591451    .72277699    .25371753    .26179618
    .68149511    .49150355    .55245542    .43581643    .94574536
    .80932557    .25083542    .21198842    .96328431    .19195103
    .08086897    .24014775    .64367995    .86508177    .26023213
    .99818840    .43843408    .38458823    .04355437    .79000334
    .58945497    .79869858    .47956466    .42636062    .70485364
    .19984298    .43762268    .31051758    .13649140    .24745358
    .50565149    .44723516    .62604173    .19474390    .93385258
    .59391388    .12793158    .66413389    .40995102    .28074620
    .80745934    .78688095    .94384994    .85280256    .33018908
    .52566658    .46310426    .42557129    .14715059    .86727976
    .08506605    .54269190    .43609520    .55454979    .52246546
    .16806210    .33180403    .12150000    .59420798    .10730171
    .67751298    .79442988    .48821383    .66874291    .26061663
    .47433762    .17345499    .63166744    .24079636    .57410409
    .94544161    .30115672    .41569074    .42768670    .36787868
    .47184482    .63766813    .27545039    .93217368    .68609919
    .31508473    .19178458    .45662838    .42237063    .99065714
    .92248669    .62696406    .47829325    .37869447    .44642518
***Vecteur trie***
    .04355437    .08086897    .08506605    .09591451    .10730171
    .12150000    .12793158    .13649140    .14715059    .16806210
    .17345499    .19178458    .19195103    .19474390    .19984298
    .21198842    .24014775    .24079636    .24745358    .25083542
    .25371753    .26023213    .26061663    .26179618    .27545039
    .28074620    .30115672    .31051758    .31508473    .33018908
    .33180403    .35801113    .36787868    .37869447    .38458823
    .40995102    .41569074    .42237063    .42557129    .42636062
    .42768670    .43581643    .43609520    .43762268    .43843408
    .44642518    .44723516    .45662838    .46310426    .47184482
    .47433762    .47829325    .47956466    .48821383    .49150355
    .50565149    .52246546    .52566658    .54269190    .55245542
    .55454979    .57410409    .58945497    .59391388    .59420798
    .62604173    .62696406    .63166744    .63766813    .64367995
    .66371453    .66413389    .66874291    .67751298    .68149511
    .68609919    .70485364    .72277699    .73250770    .78642405
    .78688095    .79000334    .79442988    .79869858    .80745934
    .80932557    .85280256    .86508177    .86727976    .89372808
    .92248669    .93217368    .93385258    .94384994    .94544161
    .94574536    .96328431    .99065714    .99818840    .99886494
<<<FIN>>>
```

```
***Vecteur original***
    .97483061    .28369742    .66684497    .97035508    .96792278
    .70159126    .95403715    .92143027    .82078073    .21892753
    .43105017    .90571566    .31300998    .04298955    .27097714
    .88679646    .84922996    .01604105    .08817240    .56558739
    .78349955    .40281095    .42122521    .85430620    .01582928
    .36351306    .78848634    .26068202    .32807260    .01604637
    .66056150    .06764733    .39456135    .80753203    .86645844
    .95193811    .83057426    .16812844    .94056498    .13319931
    .40579520    .16643809    .18057716    .88227875    .76912211
    .62689088    .11812907    .27016548    .61927291    .83212025
    .15020464    .58159524    .43160811    .85382310    .91535162
    .99407994    .97113748    .08758822    .08681751    .85033590
    .60012640    .18537832    .09018467    .28264630    .17954293
    .71095749    .77294241    .10501930    .92411436    .66876473
    .79226391    .17246472    .17342500    .43296292    .46477408
    .59701747    .46121844    .54317529    .98056737    .55378072
    .00708743    .53216234    .59398363    .52550353    .81878059
    .70190869    .48562267    .20852435    .95821124    .56627606
    .86367237    .24331235    .69336605    .80178877    .33487309
    .09442741    .14357978    .05005947    .35134005    .02846003
***Vecteur trie***
    .99407994    .98056737    .97483061    .97113748    .97035508
    .96792278    .95821124    .95403715    .95193811    .94056498
    .92411436    .92143027    .91535162    .90571566    .88679646
    .88227875    .86645844    .86367237    .85430620    .85382310
    .85033590    .84922996    .83212025    .83057426    .82078073
    .81878059    .80753203    .80178877    .79226391    .78848634
    .78349955    .77294241    .76912211    .71095749    .70190869
    .70159126    .69336605    .66876473    .66684497    .66056150
    .62689088    .61927291    .60012640    .59701747    .59398363
    .58159524    .56627606    .56558739    .55378072    .54317529
    .53216234    .52550353    .48562267    .46477408    .46121844
    .43296292    .43160811    .43105017    .42122521    .40579520
    .40281095    .39456135    .36351306    .35134005    .33487309
    .32807260    .31300998    .28369742    .28264630    .27097714
    .27016548    .26068202    .24331235    .21892753    .20852435
    .18537832    .18057716    .17954293    .17342500    .17246472
    .16812844    .16643809    .15020464    .14357978    .13319931
    .11812907    .10501930    .09442741    .09018467    .08817240
    .08758822    .08681751    .06764733    .05005947    .04298955
    .02846003    .01604637    .01604105    .01582928    .00708743
<<<FIN>>>
```

Cet algorithme a d'ailleurs été vérifié sur des rangées d'ordre *10_000*. On remarque qu'il arrivera qu'on utilise des processeurs *tri_sec* pour trier des (sous-)rangées très courtes. En pratique, il vaudrait mieux l'éviter et utiliser un autre algorithme lorsque la dimension de la rangée à trier devient trop petite. Créer une tâche est en effet une opération relativement lourde, en tous cas nettement plus coûteuse que l'appel d'une procédure.

```
***Vecteur original***
  -.01485376 -1.79578973   .35567363  .00897414  .36517601
   .16937669   .25021779   .73778288  .48547146 1.27833508
   .70694933  -.42781937  -.54154862 -.23364125 1.02546270
   .90192248  1.44050925  -.08996553 -.93132410 -1.41004539
 -1.43345223   .23423973  -.80412535 1.05766159 1.26304265
 -1.56715884  -.18324760 -1.05068113 -.11486598 -.29563038
   .98487417  -.67078831   .25268440 -.44059912 -.94826693
  1.56144656   .51691785 -1.73680031  .58659101 -.27830544
  -.39570595  -.69366809  1.00710872  .72892877 -.19676545
 -1.57115377  1.68050657  -.04916219  .08156349 2.10558845
***Vecteur trie***
   .00897414  -.01485376  -.04916219  .08156349 -.08996553
  -.11486598   .16937669  -.18324760 -.19676545 -.23364125
   .23423973   .25021779   .25268440 -.27830544 -.29563038
   .35567363   .36517601  -.39570595 -.42781937 -.44059912
   .48547146   .51691785  -.54154862  .58659101 -.67078831
  -.69366809   .70694933   .72892877  .73778288 -.80412535
   .90192248  -.93132410  -.94826693  .98487417 1.00710872
  1.02546270 -1.05068113  1.05766159 1.26304265 1.27833508
 -1.41004539 -1.43345223  1.44050925 1.56144656 -1.56715884
 -1.57115377  1.68050657 -1.73680031 -1.79578973 2.10558845
<<<FIN>>>


***Vecteur original***
  2.59995626  8.96324532  9.03828580  8.25360320   .62145975
   .02588047  3.33319738  3.06688078  2.78059747 28.51884161
 24.38263483  3.29715475 67.40504324  2.27443219 16.67633519
 10.19165421  6.89027230  6.24596405  3.28634044  1.30142809
 12.29758665  2.13071463  1.48528164  3.81878785   .58012625
 13.98322933  3.83753097  2.30017655  1.11013827  4.47858928
  9.91870337 19.94836887  6.04297437 29.14965081  3.94730066
 12.23586892 12.70803208  6.17306857 45.39585224  4.43682556
  8.21467247 17.05385085  3.60265309  4.37078628  9.56092867
  7.88168289  2.64000805  1.84755718 13.71221476  3.05911183
***Vecteur trie***
   .58012625   .62145975   .02588047  1.30142809  1.48528164
  1.84755718  1.11013827  2.64000805  2.27443219  2.30017655
  2.13071463  2.59995626  2.78059747  3.06688078  3.81878785
  3.60265309  3.94730066  3.83753097  3.33319738  3.05911183
  3.28634044  3.29715475  4.37078628  4.43682556  4.47858928
  6.04297437  6.17306857  6.24596405  6.89027230  7.88168289
  8.21467247  8.96324532  8.25360320  9.56092867  9.03828580
  9.91870337 10.19165421 12.29758665 12.70803208 12.23586892
 13.98322933 13.71221476 16.67633519 17.05385085 19.94836887
 24.38263483 28.51884161 29.14965081 45.39585224 67.40504324
<<<FIN>>>
```

Pour le reste, on notera que plusieurs langages, dont Ada, imposent que toute tâche attende, avant de se terminer, la fin de l'exécution des sous-tâches qu'elle a engendrées. Dans un tel langage, il n'aurait donc pas été nécessaire d'expliciter la synchronisation, liée aux objets du type *salle_d_attente*, à la fin des sous-tâches du type *tri_sec*.

A ce stade, on peut poursuivre l'analogie entre coroutines et paroutines.

coroutines	paroutines
coroutine	**paroutine**
activity	*task*
activate	**schedule**
return	**interrupt**

Il est tout à fait possible de réaliser des applications faisant intervenir à la fois des coroutines et des paroutines; la chose peut d'ailleurs être intéressante. Avant d'en voir des applications, il convient de préciser la manière dont sera exécutée une application comportant à la fois des tâches et des coroutines. Dans un tel système, le programme principal est toujours considéré comme une tâche mise en oeuvre par le système d'exploitation. Tant que l'application ne fait intervenir aucune coroutine, on a **current = none**. Le cas échéant, une coroutine est toujours exécutée sous contrôle d'une tâche; par définition, cette tâche est le superviseur de la coroutine. Soit *a* une coroutine, on a donc la relation :

state *a* = *attached* /> *supervisor (a)* ~ = **nothing**

Une tâche donnée peut superviser plusieurs coroutines qui sont alors attachées les unes aux autres sous la forme d'une liste bidirectionnelle. Parmi les coroutines, celle qui a été attachée en dernier lieu et qui, par conséquent, se trouve en exécution lorsque son superviseur est à l'état *running*, est l'exécutant de la tâche (fig. 65).

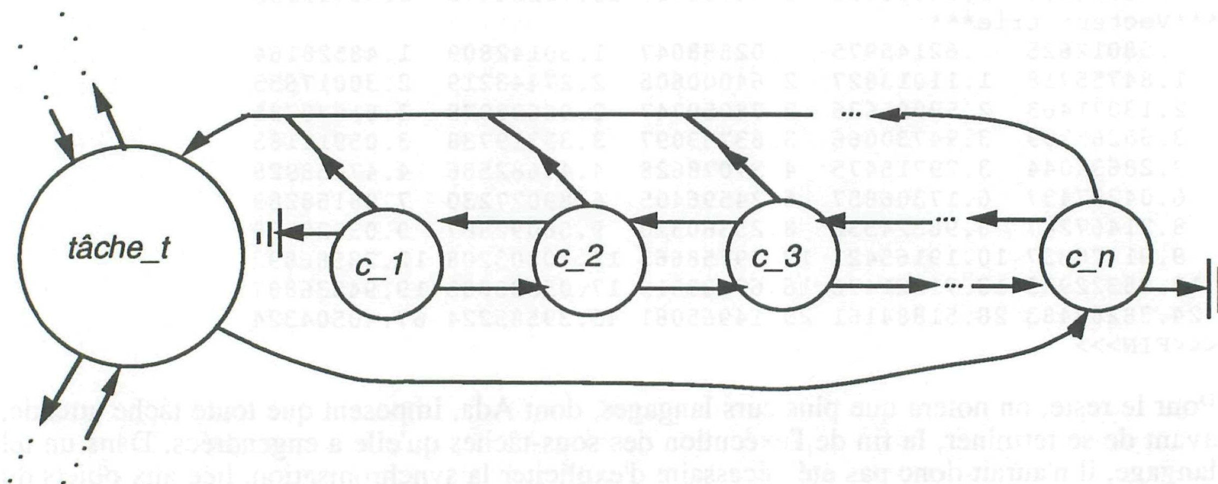


Fig. 65

Dans cette figure, la tâche *tâche_t* supervise l'exécution de chacune des coroutines c_1, c_2, ... c_n; cette dernière coroutine est l'exécutant de *tâche_t* :

executant (tâche_t) = c_n

Si, de plus, **curr_task** = *tâche_t*, son exécutant est la coroutine courante :

tâche_t = **curr_task** /> *executant (tâche_t)* = **current**

Remarques :

1. Une coroutine, tout comme une tâche, peut contrôler l'exécution d'un ensemble de procédures, fonctions et/ou fonctions d'accès.

2. Il n'est pas indispensable que le superviseur d'une coroutine donnée soit constamment la même tâche. Une coroutine détachée peut très bien être réattachée à une tâche différente de celle qui était son superviseur lors de sa phase d'activité précédente.

On va voir maintenant, au moyen d'un exemple, l'utilisation de coroutines pour implanter la notion de rendez-vous; le rendez-vous est le mécanisme de synchronisation et d'exclusion mutuelle prédéfini dans le langage Ada. Tel qu'il est défini en Ada, ce mécanisme fait intervenir, de manière asymétrique, deux tâches : une tâche réceptrice *r* et une tâche client *c*.

Pour la tâche réceptrice *r*, un lieu de rendez-vous se présente sous la forme d'une section critique de code à exécuter en exclusion mutuelle par rapport à la tâche client *c*; cette section peut dépendre d'un ensemble de paramètres formels (avec les mêmes modes de passages que les paramètres de procédures). Pour la tâche client *c*, une prise de rendez-vous a la forme d'un appel de procédure.

Lorsqu'une tâche réceptrice atteint un lieu de rendez-vous, elle commence par examiner si un client a pris le rendez-vous correspondant (chaque lieu de rendez-vous est baptisé au moyen d'un identificateur); si ce n'est pas le cas, la tâche réceptrice se met en attente. Lorsqu'une tâche client effectue une prise de rendez-vous, elle examine si la tâche réceptrice concernée a atteint un lieu de rendez-vous correspondant; dans le cas contraire, la tâche client se met en attente. Plusieurs clients peuvent chercher à prendre le même rendez-vous; dans ce cas, ils seront servis dans l'ordre dès l'élaboration des prises de rendez-vous correspondantes. Un rendez-vous a lieu lorsqu'une tâche réceptrice a atteint un lieu de rendez-vous et qu'une tâche client a pris le rendez-vous concerné. Dans ce cas, le client commence par communiquer à la tâche réceptrice les paramètres effectifs de l'énoncé de prise de rendez-vous; après cela, le client attend pendant que la tâche réceptrice élabore la section critique correspondante. Une fois cette dernière achevée, le rendez-vous est terminé : la tâche client est débloquée; la tâche réceptrice et la tâche client poursuivent leur exécution en parallèle indépendamment l'une de l'autre.

Le programme *parmat* suivant calcule une approximation de la plus grande valeur propre d'une matrice aléatoire *A* à termes positifs et du vecteur propre qui lui est associé en utilisant la méthode de la puissance. L'idée est de partir d'une approximation du vecteur propre et de la pré-multiplier, de manière répétée, par la matrice. Au bout d'un certain nombre d'itérations, ce procédé peut converger vers le vecteur propre souhaité; soit $\vec{v}$ le vecteur itéré, en cas de convergence, le quotient de Rayleigh suivant converge vers la valeur propre associée à $\vec{v}$:

$$(\vec{v}^T A \vec{v}) / (\vec{v}^T \vec{v})$$

Il est clair que le produit d'une matrice par un vecteur peut être effectué en parallèle : les composantes du vecteur produit sont calculables indépendamment les unes des autres. Le calcul de chacune des composantes du vecteur produit pourra donc être confié à une sous-tâche créée à cet effet. La tâche principale devra alors attendre que chacune de ces sous-tâches ait terminé son exécution; elle pourrait le faire au moyen d'un objet du type *salle_d_attente*; dans le cas du programme *parmat*, cette synchronisation est fait au moyen de rendez-vous. La figure 66 décrit la structure de ce programme.

```
program parmat

    integer subrange naturel subrange positif

    class verrou
    class rendez_vous

    real row vecteur row matrice
    paroutine imprimerie

    real function pro_scal
        (vecteur value u, v)
    vecteur function unité
        (vecteur value v)
    vecteur function mat_vec
        (matrice value a; vecteur value v)

     constant ordre, itérations
     matrice value a
     vecteur variable av, v
     real variable quot
```

Fig. 66

Dans ce programme *a* est une matrice carrée aléatoire de dimension *ordre*. Partant du vecteur *av* dont toutes les composantes sont égales à *1*, il est effectué *itérations* fois le calcul suivant. En *v*, le vecteur *av* est ramené à la longueur unité; le produit *av* de la matrice *a* par le vecteur *v* est ensuite élaboré. Finalement le produit scalaire des vecteurs *v* et *av* est égal au quotient de Rayleigh *quot*; son calcul termine une itération.

La classe *verrou* est bien connue.

```
parmat                                              Vax Newton Compiler 1.0
                                            Page    1
                Source listing

/* /*OLDSOURCE=USER2:[RAPIN]PARMAT.NEW*/ */
PROGRAM parmat DECLARE
  integer SUBRANGE naturel(naturel>=0)
          SUBRANGE positif(positif>0);

  CLASS verrou  VALUE moi
    ATTRIBUTE libre,fermer,ouvrir
  (*Un objet du type  verrou  est utilise pour forcer l'exclusion
    mutuelle a une section critique. Les processeurs en attente
    de pouvoir executer leur section critique seront liberes dans
    un ordre non defini a priori
  *)
  DECLARE(*verrou*)
    Boolean VARIABLE ouvert VALUE libre:=TRUE;
    (*FALSE ssi un processeur est dans sa section critique*)

    verrou EXPRESSION fermer=
    (*A executer avant d'entrer en section critique protegee par
      le verrou concerne; le resultat est le verrou concerne
    *)
      (UNTIL FALSE=:ouvert REPEAT SWITCH REPETITION; moi);

    verrou EXPRESSION ouvrir=
    (*A executer avant de quitter la section critique protegee par
      le verrou concerne; le resultat est le verrou concerne
    *)
      (ouvert:=TRUE; moi)
  DO(*verrou*)DONE;
  /* /*EJECT*/ */
```

La structure de la classe *rendez_vous* est donnée à la figure 67

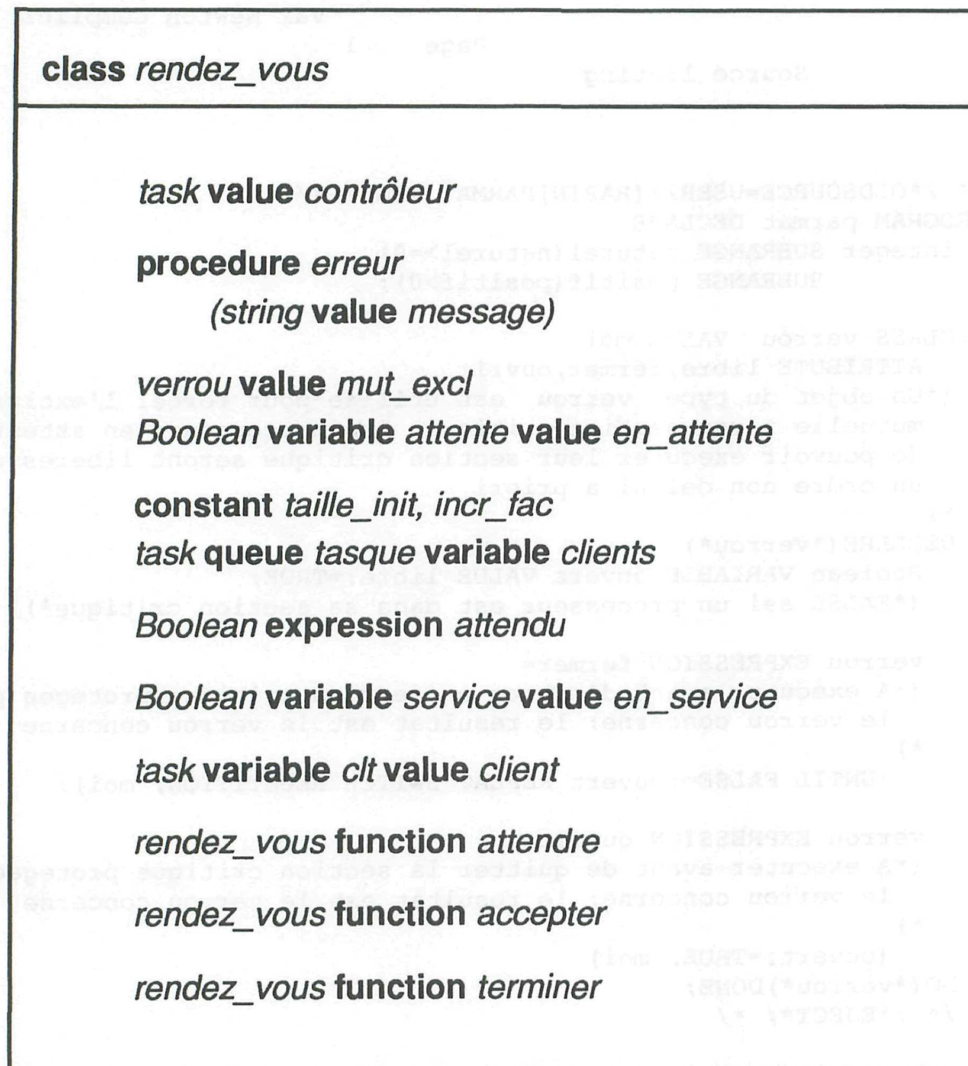


Fig. 67

```
  69    CLASS rendez_vous  VALUE moi
  73      ATTRIBUTE controleur,attendre,en_attente,attendu,
  82                accepter,en_service,client,terminer
  89    DECLARE(*rendez_vous*)
  90      task VALUE controleur=CURR_TASK;
  96      (*la tache a laquelle est incorpore cet objet; attention de
  96        s'assurer que cet objet a ete elabore avant de l'utiliser
  96      *)
  96
  96      PROCEDURE erreur(string VALUE message) DO
 104        print(line,"###Erreur dans rendez-vous###",
 110              line,___,message,
 116              line,"###Tache avortee###",line)
 122      INTERRUPT DONE;
 125
 125      verrou VALUE mut_excl=verrou;
 131
 131      Boolean VARIABLE attente VALUE en_attente:=FALSE;
 139      (*vrai ssi le controleur est en attente d'un rendez-vous*)
 139
 139      CONSTANT taille_init=20,incr_fac=1.5;
 150      task QUEUE tasque VARIABLE clients:=tasque(taille_init);
 161
 161      Boolean EXPRESSION attendu=
 165      (*vrai ssi un client attend un rendez-vous*)
 165        ~EMPTY clients;
 169
 169      Boolean VARIABLE service VALUE en_service:=FALSE;
 177      (*vrai ssi le controleur execute une section critique*)
 177
 177      task VARIABLE clt VALUE client:=NOTHING;
 185      (*invariant:  en_service == client~=NOTHING
 185
 185        le cas echeant, le client pour lequel le controleur
 185        execute une section critique
 185      *)
 185
 185      rendez_vous FUNCTION attendre
 188      (*attend que le controleur ait atteint un lieu de rendez-vous;
 188        le resultat est le rendez-vous concerne
 188
 188        condition d'emploi:  CURR_TASK~=controleur /\ CURRENT~=NONE
 188      *)
 188      DO(*attendre*)
 189        UNLESS CURR_TASK=controleur THEN
 194          UNLESS CURRENT=NONE THEN
 199            mut_excl.fermer;
 203            UNLESS en_attente THEN
 206              IF FULL clients THEN
 210                THROUGH
 211                  (tasque(CAPACITY clients*incr_fac)=:clients)
 222                  VALUE client
```

```
                REPEAT clients APPEND client REPETITION
              DONE;
              clients APPEND CURR_TASK;
              mut_excl.ouvrir INTERRUPT
            DEFAULT
              clt:=CURR_TASK;
              attente:=FALSE;
             UNTIL STATE controleur=waiting REPEAT SWITCH REPETITION;
              controleur RESTART
            DONE
          DEFAULT
          erreur("'attendre' n'est pas execute depuis une coroutine")
          DONE
        DEFAULT
          erreur("'attendre' execute par controleur")
        DONE
      TAKE moi DONE(*attendre*);

      rendez_vous FUNCTION accepter
      (*lieu de rendez-vous; le controleur se met en attente si
        aucune tache n'a demande le rendez-vous; le resultat est le
        rendez-vous concerne

        condition d'emploi: CURR_TASK=controleur EXCL en_service
      *)
      DECLARE activity VARIABLE critique DO
        IF CURR_TASK=controleur THEN
          UNLESS en_service THEN
            mut_excl.fermer;
            IF EMPTY clients THEN
              attente:=TRUE;
              mut_excl.ouvrir INTERRUPT
            DEFAULT
              clt FROM clients
            DONE;
            service:=TRUE;
            UNTIL STATE client=waiting REPEAT SWITCH REPETITION;
            HOLD critique:=executant(client) SUSPEND;
            mut_excl.ouvrir;
            ACTIVATE critique NOW
          DEFAULT
            erreur("'accepter' recursif sur meme rendez-vous")
          DONE
        DEFAULT
       erreur("'accepter' execute par tache autre que le controleur")
        DONE
      TAKE moi DONE(*accepter*);

      rendez_vous FUNCTION terminer
      (*termine section critique d'un rendez-vous; le resultat
        est le rendez-vous concerne
```

```
parmat                                          Vax Newton Compiler 1.0
                                          Page    4
              Source listing

      condition d'emploi: CURR_TASK=controleur/\en_service
    *)
    DO(*terminer*)
      IF CURR_TASK=controleur THEN
        IF en_service THEN
          mut_excl.fermer;
          service:=FALSE;
          SCHEDULE NOTHING=:clt NOW;
          mut_excl.ouvrir
        DEFAULT
          erreur("'terminer' ne termine pas un rendez-vous")
        DONE
      DEFAULT
     erreur("'terminer' execute par tache autre que le controleur")
      DONE
   TAKE moi DONE
  DO(*rendez_vous*)DONE;
  /* /*EJECT*/ */
```

Tout objet *rv* du type *rendez_vous* est à même de désigner un lieu de rendez-vous, voire plusieurs lieux de rendez-vous de même nom. L'attribut *contrôleur* dénote la tâche qui a créé l'objet correspondant; il s'agira obligatoirement de la tâche réceptrice. Dans cette dernière, l'algorithme d'un rendez-vous prendra la forme d'une déclaration de processus; la section critique correspondante sera incorporée à la partie exécutable de cette dernière : elle sera encadrée des énoncés *rv.attendre* et *rv.terminer*. Dans la partie exécutable de la tâche réceptrice, un lieu de rendez-vous est indiqué au moyen de l'énoncé *rv.accepter*. Pour prendre un rendez-vous, une tâche client crée un objet processus contenant une section critique encadrée par les énoncés *rv.attendre* et *rv.terminer*. On remarque, en particulier, que l'énoncé *rv.attendre* est exécuté par la tâche client; ceci est vérifié par le test initial **unless curr_task** = *contrôleur*. Au moyen du test **unless current = none**, il est aussi vérifié que cette primitive est exécutée depuis une coroutine (ou éventuellement une procédure subordonnée à une coroutine). On remarque l'apparition de la primitive **restart** qui est l'analogue, au niveau des tâches, de **resume**; appliquée à une tâche en attente, elle réactive cette dernière tandis que la tâche qui exécute cette clause se met en attente. Globalement, la variable *attente* permet de savoir si la tâche réceptrice *contrôleur* a atteint un lieu de rendez-vous approprié. Si ce n'est pas le cas, la tâche client se met en attente dans la queue extensible *clients*. Par contre, si la tâche réceptrice a atteint un lieu de rendez-vous, la tâche client relance cette dernière après s'être insérée dans la variable *clt* de valeur *client* et avoir mis à jour la variable *attente*.

La primitive *accepter* est donc élaborée par la tâche réceptrice lorsqu'elle atteint un lieu de rendez-vous. La variable *service*, de valeur *en_service*, est vraie lorsque la tâche réceptrice élabore une section critique protégée par le rendez-vous concerné. On remarque que l'on interdit d'accepter récursivement le même rendez-vous dans une telle section critique. On remarque que le langage Ada ne l'interdit pas formellement; cependant, le rapport de définition du langage indique qu'un tel lieu de rendez-vous récursif impliquera un blocage de la tâche réceptrice : mieux vaut donc l'interdire. Après ces tests de validité, deux cas sont possibles selon que des tâches clients aient pris le rendez-vous correspondant ou non. Dans le premier cas, le premier de ces clients est détaché de la file d'attente *clients* et inséré dans la variable *clt* afin de pouvoir être servi; dans l'autre cas, la tâche réceptrice se met en attente après avoir stocké **true** dans la variable *attente* : elle sera réveillée par la clause **restart** exécutée par le client éventuel. Dans les deux cas, la tâche réceptrice reprend le contrôle au moment où le rendez-vous peut intervenir : le client correspondant figure alors dans la variable *clt*. Après avoir mis à jour la variable *service* et s'être assuré que le client est bien en attente, il est élaboré l'énoncé **hold** *critique := executant (client)* **suspend**. L'énoncé **hold** est une primitive de manipulation de coroutines; appliqué à

une coroutine *a* attachée, l'énoncé **hold *a* suspend** la détache (on notera que cet énoncé possède un analogue **preempt** au niveau des tâches). Ainsi, au moyen de cet énoncé **hold**, la tâche réceptrice détache du client son exécutant : cet exécutant est la coroutine contenant la section critique. Après libération du verrou *mut_excl*, la tâche réceptrice fait exécuter la section critique sous son propre contrôle.

C'est donc la tâche réceptrice qui exécute la primitive *terminer* marquant la fin de la section critique. Cette primitive ne fait que stocker **false** dans la variable *service*, **nothing** dans la variable *clt* tout en réactivant la tâche contenue au préalable dans cette variable. Les actions sont faites, évidemment, sous protection du verrou *mut_excl* après contrôles de validité appropriés.

Pour cerner de plus près le fonctionnement des objets de cette classe, on peut considérer les états possibles des variables (ou valeurs) *en_attente*, *en_service*, *client* et *clients*. Quatre cas sont possibles :

<u>1e cas</u> : La tâche réceptrice n'a pas atteint un lieu de rendez-vous; aucun client n'a demandé le rendez-vous :

~ *en_attente*
~ *en_service*
empty *clients*
client = **nothing**

c'est en particulier là l'état initial.

<u>2e cas</u> : La tâche réceptrice a atteint un lieu de rendez-vous; aucun client n'a demandé le rendez-vous :

en_attente
~ *en_service*
empty *clients*
client = **nothing**
state *contrôleur* = *waiting*

<u>3e cas</u> : La tâche réceptrice n'a pas atteint un lieu de rendez-vous; un client a demandé le rendez-vous :

~ *en_attente*
~ *en_service*
~ **empty** *clients*
client = **nothing**

<u>4e cas</u> : La tâche réceptrice exécute pour un client la section critique d'un rendez-vous :

~ *en_attente*
en_service
client ~= **nothing**
state *client* = *waiting*

A noter que, dans ce dernier cas, la file d'attente *clients* peut être vide ou non selon que d'autres clients attendent ou non sur le même rendez-vous. On notera cependant que l'exécution d'un énoncé *rv.accepter* ne permet de servir qu'un seul client. Si la queue *clients* n'est pas vide au moment d'exécuter la primitive *rv.terminer*, on revient au troisième cas ci-dessus; si elle est vide, on revient au premier cas.

Cette classe permet de poursuivre la table comparative entre paroutines et coroutines.

coroutines	paroutines
resume **hold**	**restart** **preempt**

Dans cette application, toutes les sorties de résultats sont centralisées dans la paroutine *imprimerie*; cette dernière exporte quatre opérations *imprime_texte*, *imprime_scalaire*, *imprime_vecteur* et *imprime_matrice*. Ces dernières sont définies sous la forme de processus. L'idée est de permettre d'exécuter en parallèle l'impression de résultats et la suite des calculs; les impressions successives devront par contre être élaborées dans l'ordre. Pour celà, la paroutine *imprimerie* contient un rendez-vous *imprime*; sa partie exécutable ne fait qu'accepter répétitivement les rendez-vous correspondants. On remarque que les quatre primitives exportées contiennent toutes une section critique vide. C'est ceci qui permet d'assurer le parallélisme entre les opérations d'impression et la suite du traitement. Ce parallélisme implique par contre certaines contraintes aux tâches clients : pendant l'élaboration des primitives *imprime_vecteur* et *imprime_matrice*, elles ne doivent pas modifier les valeurs des composantes du vecteur ou respectivement de la matrice concernée (à défaut, l'effet de l'impression ne sera pas défini).

```
parmat                                          Vax Newton Compiler 1.0
                                                    Page     5
                   Source listing

     real ROW vecteur ROW matrice;

  PAROUTINE imprimerie
     ATTRIBUTE imprime_texte,imprime_scalaire,
               imprime_vecteur,imprime_matrice
  (*utilisee pour imprimer des valeurs reelles, des vecteurs et des
     matrices, cette paroutine est executee parallelement au
     programme principal; les impressions successives sont par contre
     sequencees.

  Attention: avant d'utiliser les procedures attributs, s'assurer que
               cette paroutine ait passe sur son premier point
               d'interruption
  *)
  DECLARE(*imprimerie*)
     rendez_vous VALUE imprime=rendez_vous;

     PROCESS imprime_texte
        (string VALUE texte)
      (*imprime, sur une ligne, la chaine donnee  texte *)
     DO(*imprime_texte*)
        imprime.attendre; imprime.terminer;
        print(line,texte,line)
     DONE(*imprime_texte*);

     PROCESS imprime_scalaire
        (real VALUE val; string VALUE texte;
         positif VALUE chp; naturel VALUE dig)
    (*imprime la valeur  val  donnee, arondie a  dig  chiffres decimaux
     dans un champ de  chp  caracteres; la valeur est imprimee sur une
     ligne precedee du  texte  donne
     *)
     DO(*imprime_scalaire*)
        imprime.attendre; imprime.terminer;
        print(line,texte,":"_,edit(val,chp,dig),line)
     DONE(*imprime_scalaire*);

     PROCESS imprime_vecteur
        (vecteur VALUE vec; string VALUE texte;
         positif VALUE chp; naturel VALUE dig)
    (*imprime les composantes du vecteur  vec  donne; chaque composante
    sera imprimee dans un champ de  chp  caracteres et arrondie a  dig
    decimales; les composantes individuelles seront precedees de leur
    indice. L'ensemble du vecteur sera precede du titre  texte

        Attention: ne pas modifier les valeurs des composantes tant que
                   l'impression n'est pas terminee
     *)
     DO(*imprime_vecteur*)
        imprime.attendre; imprime.terminer;
        print(line,#***vecteur "#,texte,#"***#,line);
        THROUGH vec INDEX k VALUE vec_k REPEAT
```

```
parmat                                          Vax Newton Compiler 1.0
                                     Page     6
               Source listing

  553         edit(k,5,0); edit(vec_k,chp,dig); line
  572       REPETITION;
  574       print("<<<FIN>>>",line)
  580     DONE(*imprime_vecteur*);
  582
  582     PROCESS imprime_matrice
  584       (matrice VALUE mat; string VALUE texte;
  593        positif VALUE chp; naturel VALUE dig; positif VALUE quant)
  605      (*imprime les composantes de la matrice  mat  donnee; chaque
  605       composante est imprimee dans un champ de  chp  caracteres
  605       arrondie a  dig  decimales. Chaque ligne de la matrice est
  605       precedee d'un texte indiquant son indice; les elements
  605       d'une meme ligne sont imprimes a raison de  quant  par
  605       ligne d'impression: le premier element de chaque ligne est
  605       precede de son indice. L'ensemble de la matrice est precedee
  605       du titre  texte
  605
  605       Attention:  ne pas modifier la valeur des composantes de la
  605                   matrice pendant l'impression
  605     *)
  605     DO(*imprime_matrice*)
  606       imprime.attendre; imprime.terminer;
  614       print(line,#***matrice "#,texte,#"***#);
  625       THROUGH mat INDEX j VALUE mat_j REPEAT
  632         print(line,___"ligne:"_,edit(j,5,0));
  650         THROUGH mat_j INDEX k VALUE mat_jk REPEAT
  657           IF (k-LOW mat_j)\quant=0 THEN
  669             line; print(___); edit(k,5,0)
  684           DONE;
  686           edit(mat_jk,chp,dig)
  694         REPETITION
  695       REPETITION;
  697       print(line,"<<<FIN>>>",line)
  705     DONE(*imprime_matrice*)
  706   DO(*imprimerie*)INTERRUPT
  708     LOOP imprime.accepter REPETITION
  713   DONE(*imprimerie*);
  715   /* /*EJECT*/ */
```

```
real FUNCTION pro_scal
  (vecteur VALUE u,v)
(*le produit scalaire des deux vecteurs donne*)
DECLARE real VARIABLE s:=0 DO
  UNLESS LOW u=LOW v/\HIGH u=HIGH v THEN
   print(line,"###produit scalaire de vecteurs incompatibles###")
  INTERRUPT DONE;
  THROUGH u INDEX k VALUE u_k REPEAT
   s:=s+u_k*v[k]
  REPETITION
TAKE s DONE(*pro_scal*);

vecteur FUNCTION unite
  (vecteur VALUE v)
(*un vecteur unite proportionnel a  v *)
DECLARE real VALUE lg=sqrt(pro_scal(v,v)) DO
  IF lg=0 THEN
    print(line,"###le vecteur nul ne peut etre normalise###")
  INTERRUPT DONE
TAKE
  THROUGH
    vecteur(LOW v TO HIGH v) INDEX k:=v[k]/lg
  REPETITION
DONE(*unite*);

vecteur FUNCTION mat_vec
  (matrice VALUE a; vecteur VALUE v)
(*le produit de la matrice  a  par le vecteur  v *)
DECLARE(*mat_vec*)
  vecteur VALUE av=vecteur(LOW a TO HIGH a);

  rendez_vous VALUE stocke_elt=rendez_vous;

  PROCESS stocker
    (real VALUE elt; integer VALUE pos)
  DO(*stocker*)
    stocke_elt.attendre;
    av[pos]:=elt;
    stocke_elt.terminer
  DONE(*stocker*);

  PROCESSOR calculer
    (integer VALUE pos)
  DECLARE real VALUE elt=pro_scal(a[pos],v) DO
    stocker(elt,pos)
  DONE(*calculer*)
DO(*mat_vec*)
  (*calcule en parallele les elements du vecteur produit*)
  THROUGH a INDEX k REPEAT calculer(k) REPETITION;
  (*attend que tous les resultats soient presents*)
  THROUGH av REPEAT stocke_elt.accepter REPETITION
TAKE av DONE(*mat_vec*);
```

```
   946      /* /*EJECT*/ */
```

La fonction *mat_vec* calcule, en parallèle, le produit d'une matrice par un vecteur. Pour chaque composante du vecteur produit, elle lance une tâche du type processeur *calculer* en lui fournissant, comme paramètre, l'indice *pos* de la composante qu'elle est chargée d'évaluer. Après avoir calculé le produit scalaire approprié *elt*, chacune de ces tâches subordonnées la fait stocker dans le vecteur résultat *av* par l'intermédiaire de l'énoncé de prise de rendez-vous *stocker (elt, pos)* : le stockage de la valeur elt dans l'élément *av[pos]* est effectué en section critique.

Après avoir lancé l'évaluation parallèle des éléments du vecteur produit, la tâche principale accepte, au moyen de l'énoncé **through** *av* **repeat** *stocke_elt.accepter* **repetition**, les rendez-vous impliqués par ces stockages; vu l'indéterminisme lié au parallélisme, l'ordre dans lequel ces stockages interviendront ne correspond pas nécessairement à l'ordre séquentiel impliqué par la clause **through** *av* : cet itérateur ne fait que s'assurer que le nombre de rendez-vous acceptés soit correct.

```
parmat                                              Vax Newton Compiler 1.0
                                        Page    9
                   Source listing

    CONSTANT ordre=5,
             iterations=5;

    matrice VALUE a=
      THROUGH
        matrice(1 TO ordre):=
          THROUGH vecteur(1 TO ordre):=random REPETITION
      REPETITION;

    vecteur VARIABLE av:=
      THROUGH vecteur(1 TO ordre):=1 REPETITION;
    vecteur VARIABLE v;

    real VARIABLE quot
  DO(*parmat*)
    (*s'assure que l'imprimerie est prete*)
    UNTIL STATE imprimerie=waiting REPEAT SWITCH REPETITION;
    SCHEDULE imprimerie NOW;
    (*imprime la matrice donnee*)
    imprime_matrice(a,"aleatoire donnee",12,8,5);
    (*effectue les iterations requises*)
    FOR integer VALUE k FROM 1 TO iterations REPEAT
      imprime_scalaire(k,"iteration",5,0);
      v:=unite(av);
      imprime_vecteur(v,"initial",15,8);
      av:=mat_vec(a,v);
      imprime_vecteur(av,"transforme",15,8);
      quot:=pro_scal(v,av);
      imprime_scalaire(quot,"quotient de Rayleigh",18,10)
    REPETITION;
    imprime_texte("<<<FIN APPLICATION VECTORIELLE>>>")
  DONE(*parmat*)

  ****  No messages were issued  ****
```

La partie exécutable du programme ne nécessite que peu de commentaires. On remarque cependant qu'avant d'entreprendre les calculs, on s'assure que la paroutine *imprimerie* ait atteint le point d'interruption au début de sa partie exécutable. La chose est importante dès le moment où il faut s'assurer que l'objet rendez-vous *imprime* ait été correctement créé et initialisé avant de pouvoir utiliser les primitives exportées de cette paroutine.

```
***matrice "aleatoire donnee"***
   ligne:     1
      1  .68221327   .89547662   .65853325   .57439188   .58088395
   ligne:     2
      1  .17131704   .28845677   .38158395   .78945317   .18931101
   ligne:     3
      1  .70552214   .45009981   .28345362   .56272916   .99408803
   ligne:     4
      1  .22667535   .05035978   .90803902   .69956268   .91377145
   ligne:     5
      1  .27925209   .18045536   .45950768   .83385605   .23276486
<<<FIN>>>

iteration:     1

***vecteur "initial"***
    1       .44721360
    2       .44721360
    3       .44721360
    4       .44721360
    5       .44721360
<<<FIN>>>

***vecteur "transforme"***
    1     1.51672445
    2       .81398328
    3     1.33980397
    4     1.25148623
    5       .88809287
<<<FIN>>>

quotient de Rayleigh:          2.5983515983

iteration:     2

***vecteur "initial"***
    1       .56869521
    2       .30520270
    3       .50235895
    4       .46924425
    5       .33299006
<<<FIN>>>

***vecteur "transforme"***
    1     1.45505205
    2       .81064213
    3     1.27607306
    4     1.23298324
    5       .91351313
<<<FIN>>>

quotient de Rayleigh:          2.5986991155

iteration:     3

***vecteur "initial"***
    1       .55974167
    2       .31184464
    3       .49089053
    4       .47431437
    5       .35141792
<<<FIN>>>
```

```
***vecteur "transforme"***
    1      1.46095586
    2       .81413920
    3      1.29066692
    4      1.24126011
    5       .91545868
<<<FIN>>>

quotient de Rayleigh:          2.6156750768

iteration:      4

***vecteur "initial"***
    1       .55853585
    2       .31125234
    3       .49343294
    4       .47454429
    5       .34998764
<<<FIN>>>

***vecteur "transforme"***
    1      1.46057834
    2       .81464265
    3      1.28897780
    4      1.24211945
    5       .91604213
<<<FIN>>>

quotient de Rayleigh:          2.6154130030

iteration:      5

***vecteur "initial"***
    1       .55845019
    2       .31147754
    3       .49283895
    4       .47492272
    5       .35024749
<<<FIN>>>

***vecteur "transforme"***
    1      1.46069872
    2       .81481423
    3      1.28932164
    4      1.24207420
    5       .91616195
<<<FIN>>>

quotient de Rayleigh:          2.6157244182

<<<FIN APPLICATION VECTORIELLE>>>
```

On notera que sur la matrice d'ordre cinq présentée ici comme exemple, le parallélisme n'a guère le temps d'intervenir. Par contre, ce programme a été expérimenté sur une matrice d'ordre mille, suite à une modification de la constante *ordre*; l'expérience a montré que, dans ce cas, le système de gestion des tâches a fortement été mis à contribution.

Par l'intermédiaire de coroutines, deux tâches peuvent communiquer de manière asynchrone. Plus spécifiquement, par l'intermédiaire de messages ad-hoc, une tâche peut demander l'exécution prioritaire d'une coroutine par une autre tâche. Le programme *meth* suivant en illustre le principe.

```
meth                                          Vax Newton Compiler 1.0
                                         Page     1
                    Source listing


     1  /* /*OLDSOURCE=USER2:[RAPIN]METH.NEW*/ */
     1  PROGRAM meth DECLARE
     4
     4  PAROUTINE par
     6    METHOD mess
     8  DECLARE
     9    PROCESS mess(integer VALUE k) DO
    17     RETURN
    18     print(line,"$$$Message no:"_,edit(k,8,0),"$$$",line)
    38    DONE;
    40
    40    integer VARIABLE j:=17
    45  DO(*par*)
    46    LOOP
    47      IF j=17 THEN line DEFAULT print(_) DONE;
    60      edit(j,3,0);
    69      j:=IF EVEN j THEN j%2 DEFAULT 3*j-1 DONE
    85    REPETITION
    86  DONE(*par*);
    88
    88    integer VARIABLE i
    91  DO(*meth*)
    92    randomize;
    94    LOOP
    95      i:=CEIL(10000*poisson);
   104      FOR integer FROM 1 TO i REPETITION;
   112      mess(i) SEND
   117    REPETITION
   118  DONE(*meth*)

        ****  No messages were issued  ****
```

Ci-après, il est donné le début d'une exécution spécifique de ce programme.

```
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17
$$$Message no:       2827$$$

$$$Message no:        682$$$
  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17
$$$Message no:       8245$$$
  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
```

```
 17  50  25  74  37 110
$$$Message no:     2698$$$

$$$Message no:      1088$$$
  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182
$$$Message no:     7674$$$
  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
```

```
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82
$$$Message no:      20999$$$
  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164
$$$Message no:        998$$$
  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
 17  50  25  74  37 110  55 164  82  41 122  61 182  91 272 136  68  34
```

Ce programme implique l'exécution concurrente de deux tâches, le programme principal et la paroutine *par*. On remarque dans cette paroutine la définition de processus *mess* spécifiée comme méthode dans l'entête de la paroutine. D'une manière générale, une déclaration de processeur ou de paroutine peut comporter une spécification de méthodes de la forme :

method suite_d_identificateurs_de_méthodes

Une spécification de méthodes est analogue à une spécification d'attributs. Tous les identificateurs de méthode cités dans cette liste sont d'ailleurs automatiquement considérés comme des attributs de la paroutine ou des processeurs correspondants; il n'y a donc pas lieu de les répéter dans la liste attribut de la paroutine ou de la déclaration de processeur. Chaque identificateur de méthode doit faire l'objet d'une déclaration de coroutine ou de processus dans la partie déclarative de la déclaration de paroutine ou de processeur correspondante.

Toute coroutine ou tout objet processus spécifié comme méthode peut être exécuté de manière prioritaire par la tâche à l'intérieur de laquelle elle est déclarée suite à l'envoi d'un message par une autre tâche. L'envoi d'un message est commandé par la clause postfixée **send**. Une telle

clause d'envoi de message ne peut porter que sur une coroutine ou un objet processus détaché et spécifié comme méthode. L'effet est de faire réattacher cette coroutine à la tâche contenant sa déclaration et de la faire exécuter de manière prioritaire par cette dernière.

Il est maintenant facile de comprendre l'effet du programme *meth*. En fonctionnement normal, la paroutine *par* fait imprimer de manière répétitive la même liste de dix-huit nombres entiers *17, 50, 25, 74, 37, 110, 55, 164, 82, 41, 122, 61, 182, 91, 272, 136, 68* et *34*.

A intervalles irréguliers aléatoires, le programme principal envoie le message *mess(i)* **send** qui fait imprimer, séance tenante, par la paroutine *par* le numéro *i* encadré d'un texte ad-hoc. En général, plus le numéro *i* du message est grand, plus l'intervalle de temps écoulé depuis le message précédent est long.

Lorsque plusieurs messages sont envoyés à la même tâche, ceux-ci seront pris en charge dans l'ordre de leur arrivée. Pour assurer la priorité de traitement des messages, ces derniers sont attachés à leur superviseur en une liste distincte de celle des autres coroutines. La figure 68 présente le schéma d'exécution d'une tâche à laquelle sont attachées des coroutines "normales" et des méthodes.

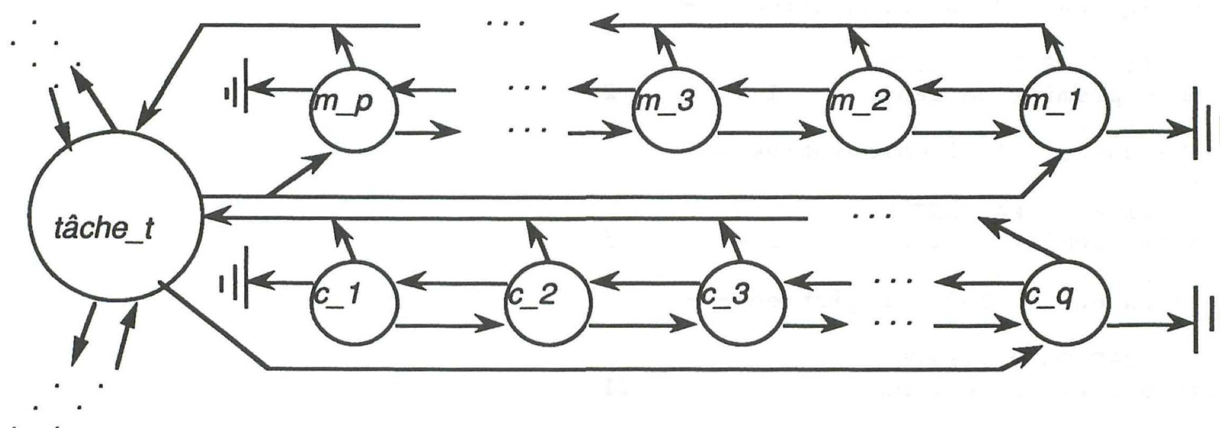


Fig. 68

Dans cette figure, la tâche *tâche_t* supervise l'exécution des coroutines "normales" *c_1, c_2, ... c_q* attachées dans cet ordre et des messages *m_1, m_2, ... m_p* envoyés dans cet ordre. L'exécutant de la tâche est la méthode *m_1*; si un nouveau message venait à être envoyé, il serait inséré dans la liste des méthodes après *m_p*. Une fois tous les messages traités, l'exécution reprendra à la coroutine "normale" *c_q*.

Une coroutine spécifiée comme méthode peut tout à fait être utilisée comme coroutine "normale" (la réciproque n'est par contre pas vraie). Ce système permet d'exprimer, de manière naturelle, un système d'exploitation. Ce dernier sera programmé sous la forme d'une tâche; les méthodes représenteront les différentes formes de requêtes susceptibles de lui être faites : ces dernières peuvent être aussi bien des commandes d'utilisateur que des requêtes internes (avertir le système qu'une opération de lecture ou d'écriture est achevée, amener une page d'informations en mémoire centrale, ...). Exécutées comme méthodes, dans l'ordre de leur arrivée, ces requêtes seront enregistrées et le minimum d'action qu'il est nécessaire de faire rapidement sera accompli; elles rendront ensuite le contrôle au système d'exploitation en se détachant. Pour le reste, le système d'exploitation fera exécuter les requêtes, comme coroutines "normales", en fonction des ressources disponibles et de critères de priorité définie de manière à assurer un service optimum.

A titre illustratif, on va montrer un système générateur de nombres premiers. A tout moment, ce système peut être interrogé par l'utilisateur; ce dernier lui transmet alors une valeur entière

positive : le système doit répondre en indiquant le nombre premier du rang correspondant. Ainsi, si l'utilisateur envoie l'entier cinq, le système doit retourner le cinquième nombre premier, c'est-à-dire onze.

Un tel système peut être construit au moyen de deux tâches. La première est le générateur de nombres premiers proprement dit. Au moyen d'un algorithme ad-hoc, cette tâche crée une table (une rangée) extensible dans laquelle elle enregistre les nombres premiers successifs. La deuxième tâche est responsable d'accepter les requêtes de l'utilisateur, de les mettre en forme et de les transmettre à la tâche génératrice. Lorsqu'il reçoit un tel message, le générateur va examiner si le nombre premier requis a déjà été créé; si c'est le cas, il le retoune immédiatement à l'utilisateur. Dans le cas contraire, la requête est mise en attente dans une queue de priorité ordonnée en fonction des rangs croissants des nombres premiers demandés (après qu'un message approprié soit retourné à l'utilisateur). Les requêtes en suspens sont débloquées dès que le nombre premier correspondant est connu. A la fin de l'utilisation du système, ce dernier imprimera la liste des nombres premiers qu'il a effectivement calculés après avoir attendu, le cas échéant, que toutes les requêtes en suspens aient été satisfaites. Cette impression finale est réalisée par la tâche génératrice suite à l'envoi d'un message de l'autre tâche.

Le listage suivant montre un exemple d'utilisation du système dans lequel il a été formé les deux mille premiers nombres premiers.

```
--->Requete    2000 enregistree<---

   PRIERE DE PATIENTER
Nombre premier de rang        1 =      2

--->Requete    1000 enregistree<---

   PRIERE DE PATIENTER
Nombre premier de rang        2 =      3

--->Requete     500 enregistree<---

   PRIERE DE PATIENTER
Nombre premier de rang        5 =     11

--->Requete     200 enregistree<---

   PRIERE DE PATIENTER

--->Requete      10 enregistree<---

   PRIERE DE PATIENTER

--->Requete     100 enregistree<---

   PRIERE DE PATIENTER

--->Requete      20 enregistree<---

   PRIERE DE PATIENTER

--->Requete      50 enregistree<---

   PRIERE DE PATIENTER
Nombre premier de rang       10 =     29
Nombre premier de rang       20 =     71
Nombre premier de rang       50 =    229
Nombre premier de rang      100 =    541
Nombre premier de rang      200 =   1223
Nombre premier de rang      500 =   3571
Nombre premier de rang     1000 =   7919
Nombre premier de rang     2000 =  17389
```

Liste des nombres premiers obtenus

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541
547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863
877	881	883	887	907	911	919	929	937	941
947	953	967	971	977	983	991	997	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231	1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451
1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
1523	1531	1543	1549	1553	1559	1567	1571	1579	1583
1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
1663	1667	1669	1693	1697	1699	1709	1721	1723	1733
1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
1823	1831	1847	1861	1867	1871	1873	1877	1879	1889
1901	1907	1913	1931	1933	1949	1951	1973	1979	1987
1993	1997	1999	2003	2011	2017	2027	2029	2039	2053
2063	2069	2081	2083	2087	2089	2099	2111	2113	2129
2131	2137	2141	2143	2153	2161	2179	2203	2207	2213
2221	2237	2239	2243	2251	2267	2269	2273	2281	2287
2293	2297	2309	2311	2333	2339	2341	2347	2351	2357
2371	2377	2381	2383	2389	2393	2399	2411	2417	2423
2437	2441	2447	2459	2467	2473	2477	2503	2521	2531
2539	2543	2549	2551	2557	2579	2591	2593	2609	2617
2621	2633	2647	2657	2659	2663	2671	2677	2683	2687
2689	2693	2699	2707	2711	2713	2719	2729	2731	2741
2749	2753	2767	2777	2789	2791	2797	2801	2803	2819
2833	2837	2843	2851	2857	2861	2879	2887	2897	2903
2909	2917	2927	2939	2953	2957	2963	2969	2971	2999
3001	3011	3019	3023	3037	3041	3049	3061	3067	3079
3083	3089	3109	3119	3121	3137	3163	3167	3169	3181
3187	3191	3203	3209	3217	3221	3229	3251	3253	3257
3259	3271	3299	3301	3307	3313	3319	3323	3329	3331
3343	3347	3359	3361	3371	3373	3389	3391	3407	3413
3433	3449	3457	3461	3463	3467	3469	3491	3499	3511
3517	3527	3529	3533	3539	3541	3547	3557	3559	3571
3581	3583	3593	3607	3613	3617	3623	3631	3637	3643
3659	3671	3673	3677	3691	3697	3701	3709	3719	3727
3733	3739	3761	3767	3769	3779	3793	3797	3803	3821
3823	3833	3847	3851	3853	3863	3877	3881	3889	3907
3911	3917	3919	3923	3929	3931	3943	3947	3967	3989
4001	4003	4007	4013	4019	4021	4027	4049	4051	4057
4073	4079	4091	4093	4099	4111	4127	4129	4133	4139
4153	4157	4159	4177	4201	4211	4217	4219	4229	4231
4241	4243	4253	4259	4261	4271	4273	4283	4289	4297
4327	4337	4339	4349	4357	4363	4373	4391	4397	4409
4421	4423	4441	4447	4451	4457	4463	4481	4483	4493

4507	4513	4517	4519	4523	4547	4549	4561	4567	4583
4591	4597	4603	4621	4637	4639	4643	4649	4651	4657
4663	4673	4679	4691	4703	4721	4723	4729	4733	4751
4759	4783	4787	4789	4793	4799	4801	4813	4817	4831
4861	4871	4877	4889	4903	4909	4919	4931	4933	4937
4943	4951	4957	4967	4969	4973	4987	4993	4999	5003
5009	5011	5021	5023	5039	5051	5059	5077	5081	5087
5099	5101	5107	5113	5119	5147	5153	5167	5171	5179
5189	5197	5209	5227	5231	5233	5237	5261	5273	5279
5281	5297	5303	5309	5323	5333	5347	5351	5381	5387
5393	5399	5407	5413	5417	5419	5431	5437	5441	5443
5449	5471	5477	5479	5483	5501	5503	5507	5519	5521
5527	5531	5557	5563	5569	5573	5581	5591	5623	5639
5641	5647	5651	5653	5657	5659	5669	5683	5689	5693
5701	5711	5717	5737	5741	5743	5749	5779	5783	5791
5801	5807	5813	5821	5827	5839	5843	5849	5851	5857
5861	5867	5869	5879	5881	5897	5903	5923	5927	5939
5953	5981	5987	6007	6011	6029	6037	6043	6047	6053
6067	6073	6079	6089	6091	6101	6113	6121	6131	6133
6143	6151	6163	6173	6197	6199	6203	6211	6217	6221
6229	6247	6257	6263	6269	6271	6277	6287	6299	6301
6311	6317	6323	6329	6337	6343	6353	6359	6361	6367
6373	6379	6389	6397	6421	6427	6449	6451	6469	6473
6481	6491	6521	6529	6547	6551	6553	6563	6569	6571
6577	6581	6599	6607	6619	6637	6653	6659	6661	6673
6679	6689	6691	6701	6703	6709	6719	6733	6737	6761
6763	6779	6781	6791	6793	6803	6823	6827	6829	6833
6841	6857	6863	6869	6871	6883	6899	6907	6911	6917
6947	6949	6959	6961	6967	6971	6977	6983	6991	6997
7001	7013	7019	7027	7039	7043	7057	7069	7079	7103
7109	7121	7127	7129	7151	7159	7177	7187	7193	7207
7211	7213	7219	7229	7237	7243	7247	7253	7283	7297
7307	7309	7321	7331	7333	7349	7351	7369	7393	7411
7417	7433	7451	7457	7459	7477	7481	7487	7489	7499
7507	7517	7523	7529	7537	7541	7547	7549	7559	7561
7573	7577	7583	7589	7591	7603	7607	7621	7639	7643
7649	7669	7673	7681	7687	7691	7699	7703	7717	7723
7727	7741	7753	7757	7759	7789	7793	7817	7823	7829
7841	7853	7867	7873	7877	7879	7883	7901	7907	7919
7927	7933	7937	7949	7951	7963	7993	8009	8011	8017
8039	8053	8059	8069	8081	8087	8089	8093	8101	8111
8117	8123	8147	8161	8167	8171	8179	8191	8209	8219
8221	8231	8233	8237	8243	8263	8269	8273	8287	8291
8293	8297	8311	8317	8329	8353	8363	8369	8377	8387
8389	8419	8423	8429	8431	8443	8447	8461	8467	8501
8513	8521	8527	8537	8539	8543	8563	8573	8581	8597
8599	8609	8623	8627	8629	8641	8647	8663	8669	8677
8681	8689	8693	8699	8707	8713	8719	8731	8737	8741
8747	8753	8761	8779	8783	8803	8807	8819	8821	8831
8837	8839	8849	8861	8863	8867	8887	8893	8923	8929
8933	8941	8951	8963	8969	8971	8999	9001	9007	9011
9013	9029	9041	9043	9049	9059	9067	9091	9103	9109
9127	9133	9137	9151	9157	9161	9173	9181	9187	9199
9203	9209	9221	9227	9239	9241	9257	9277	9281	9283
9293	9311	9319	9323	9337	9341	9343	9349	9371	9377
9391	9397	9403	9413	9419	9421	9431	9433	9437	9439
9461	9463	9467	9473	9479	9491	9497	9511	9521	9533
9539	9547	9551	9587	9601	9613	9619	9623	9629	9631
9643	9649	9661	9677	9679	9689	9697	9719	9721	9733
9739	9743	9749	9767	9769	9781	9787	9791	9803	9811
9817	9829	9833	9839	9851	9857	9859	9871	9883	9887
9901	9907	9923	9929	9931	9941	9949	9967	9973	10007
10009	10037	10039	10061	10067	10069	10079	10091	10093	10099
10103	10111	10133	10139	10141	10151	10159	10163	10169	10177

10181	10193	10211	10223	10243	10247	10253	10259	10267	10271
10273	10289	10301	10303	10313	10321	10331	10333	10337	10343
10357	10369	10391	10399	10427	10429	10433	10453	10457	10459
10463	10477	10487	10499	10501	10513	10529	10531	10559	10567
10589	10597	10601	10607	10613	10627	10631	10639	10651	10657
10663	10667	10687	10691	10709	10711	10723	10729	10733	10739
10753	10771	10781	10789	10799	10831	10837	10847	10853	10859
10861	10867	10883	10889	10891	10903	10909	10937	10939	10949
10957	10973	10979	10987	10993	11003	11027	11047	11057	11059
11069	11071	11083	11087	11093	11113	11117	11119	11131	11149
11159	11161	11171	11173	11177	11197	11213	11239	11243	11251
11257	11261	11273	11279	11287	11299	11311	11317	11321	11329
11351	11353	11369	11383	11393	11399	11411	11423	11437	11443
11447	11467	11471	11483	11489	11491	11497	11503	11519	11527
11549	11551	11579	11587	11593	11597	11617	11621	11633	11657
11677	11681	11689	11699	11701	11717	11719	11731	11743	11777
11779	11783	11789	11801	11807	11813	11821	11827	11831	11833
11839	11863	11867	11887	11897	11903	11909	11923	11927	11933
11939	11941	11953	11959	11969	11971	11981	11987	12007	12011
12037	12041	12043	12049	12071	12073	12097	12101	12107	12109
12113	12119	12143	12149	12157	12161	12163	12197	12203	12211
12227	12239	12241	12251	12253	12263	12269	12277	12281	12289
12301	12323	12329	12343	12347	12373	12377	12379	12391	12401
12409	12413	12421	12433	12437	12451	12457	12473	12479	12487
12491	12497	12503	12511	12517	12527	12539	12541	12547	12553
12569	12577	12583	12589	12601	12611	12613	12619	12637	12641
12647	12653	12659	12671	12689	12697	12703	12713	12721	12739
12743	12757	12763	12781	12791	12799	12809	12821	12823	12829
12841	12853	12889	12893	12899	12907	12911	12917	12919	12923
12941	12953	12959	12967	12973	12979	12983	13001	13003	13007
13009	13033	13037	13043	13049	13063	13093	13099	13103	13109
13121	13127	13147	13151	13159	13163	13171	13177	13183	13187
13217	13219	13229	13241	13249	13259	13267	13291	13297	13309
13313	13327	13331	13337	13339	13367	13381	13397	13399	13411
13417	13421	13441	13451	13457	13463	13469	13477	13487	13499
13513	13523	13537	13553	13567	13577	13591	13597	13613	13619
13627	13633	13649	13669	13679	13681	13687	13691	13693	13697
13709	13711	13721	13723	13729	13751	13757	13759	13763	13781
13789	13799	13807	13829	13831	13841	13859	13873	13877	13879
13883	13901	13903	13907	13913	13921	13931	13933	13963	13967
13997	13999	14009	14011	14029	14033	14051	14057	14071	14081
14083	14087	14107	14143	14149	14153	14159	14173	14177	14197
14207	14221	14243	14249	14251	14281	14293	14303	14321	14323
14327	14341	14347	14369	14387	14389	14401	14407	14411	14419
14423	14431	14437	14447	14449	14461	14479	14489	14503	14519
14533	14537	14543	14549	14551	14557	14561	14563	14591	14593
14621	14627	14629	14633	14639	14653	14657	14669	14683	14699
14713	14717	14723	14731	14737	14741	14747	14753	14759	14767
14771	14779	14783	14797	14813	14821	14827	14831	14843	14851
14867	14869	14879	14887	14891	14897	14923	14929	14939	14947
14951	14957	14969	14983	15013	15017	15031	15053	15061	15073
15077	15083	15091	15101	15107	15121	15131	15137	15139	15149
15161	15173	15187	15193	15199	15217	15227	15233	15241	15259
15263	15269	15271	15277	15287	15289	15299	15307	15313	15319
15329	15331	15349	15359	15361	15373	15377	15383	15391	15401
15413	15427	15439	15443	15451	15461	15467	15473	15493	15497
15511	15527	15541	15551	15559	15569	15581	15583	15601	15607
15619	15629	15641	15643	15647	15649	15661	15667	15671	15679
15683	15727	15731	15733	15737	15739	15749	15761	15767	15773
15787	15791	15797	15803	15809	15817	15823	15859	15877	15881
15887	15889	15901	15907	15913	15919	15923	15937	15959	15971
15973	15991	16001	16007	16033	16057	16061	16063	16067	16069
16073	16087	16091	16097	16103	16111	16127	16139	16141	16183
16187	16189	16193	16217	16223	16229	16231	16249	16253	16267

16273	16301	16319	16333	16339	16349	16361	16363	16369	16381
16411	16417	16421	16427	16433	16447	16451	16453	16477	16481
16487	16493	16519	16529	16547	16553	16561	16567	16573	16603
16607	16619	16631	16633	16649	16651	16657	16661	16673	16691
16693	16699	16703	16729	16741	16747	16759	16763	16787	16811
16823	16829	16831	16843	16871	16879	16883	16889	16901	16903
16921	16927	16931	16937	16943	16963	16979	16981	16987	16993
17011	17021	17027	17029	17033	17041	17047	17053	17077	17093
17099	17107	17117	17123	17137	17159	17167	17183	17189	17191
17203	17207	17209	17231	17239	17257	17291	17293	17299	17317
17321	17327	17333	17341	17351	17359	17377	17383	17387	17389

La figure 69 montre la structure du programme *premiers*.

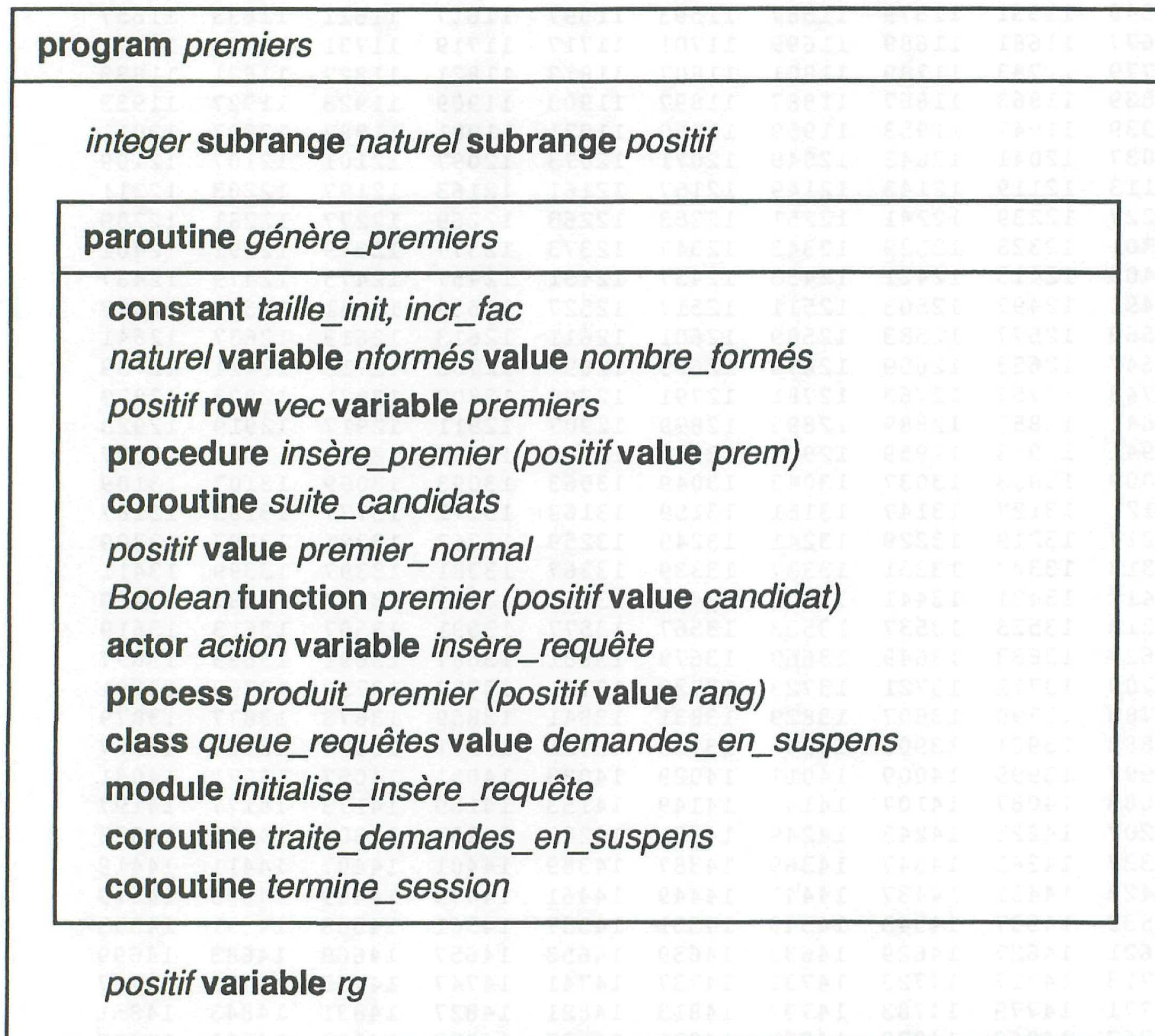


Fig. 69

Le gros du système est inclus dans la paroutine *génère_premiers*. Cette paroutine exporte comme messages le type processus *produit_premier* et la coroutine *termine_session*.

```
premiers                                            Vax Newton Compiler 0.2c16
                                                  Page    1
                   Source listing


     1  /* /*OLDSOURCE=USER2:[RAPIN]PREMIERS.NEW*/ */
     1  PROGRAM premiers DECLARE
     4
     4  integer SUBRANGE naturel(naturel>=0)
    12          SUBRANGE positif(positif>0);
    20
    20  PAROUTINE genere_premiers
    22    ATTRIBUTE nombre_formes
    24    METHOD produit_premier,termine_session
    28  DECLARE(*genere_premiers*)
    29    CONSTANT taille_init=1000,incr_fac=1.5;
    40    naturel VARIABLE nformes VALUE nombre_formes:=0;
    48    (*le nombre de valeurs premieres effectivement formees*)
    48
    48    positif ROW vec VARIABLE premiers:=vec(1 TO taille_init);
    61
    61    PROCEDURE insere_premier(positif VALUE prem) DO
    69    (*insere dans la liste le nombre premier  prem *)
    69      UNLESS nombre_formes<HIGH premiers THEN
    75        DECLARE
    76          vec VALUE nouv_premiers=vec(1 TO nombre_formes*incr_fac)
    88        DO
    89          THROUGH
    90            premiers  INDEX k  VALUE premiers_k
    95          REPEAT nouv_premiers[k]:=premiers_k REPETITION;
   104          nouv_premiers=:premiers
   107        DONE(*DECLARE nouv_premiers*)
   108      DONE(*UNLESS nombre_formes<HIGH premiers*);
   110      premiers[SUCC nombre_formes]:=prem;
   118      naturel[SUCC nformes]=:nformes
   125    DONE(*insere_premier*);
   127
   127  COROUTINE suite_candidats
   129    ATTRIBUTE premiers_speciaux,courant,prochain
   135  (*genere comme candidats a la primalite les entiers positifs
   135    non multiples de 2, 3 et 5 apres avoir produit ces trois
   135    premiers nombres premiers
   135  *)
   135  DECLARE(*suite_candidats*)
   136    CONSTANT premiers_speciaux=3;
   141    (*nombre de nombre premiers dont on a elimine les multiples*)
   141
   141    positif VARIABLE cand VALUE courant:=(RETURN 2);
   152    (*le dernier candidat produit par le systeme*)
   152
   152    positif EXPRESSION prochain=
   156    (*produit le prochain candidat a la primalite*)
   156      (ACTIVATE suite_candidats NOW; courant);
   164
   164    naturel VARIABLE k:=0
   169  DO(*suite_candidats*)RETURN
   171    cand:=3 RETURN
```

```
  cand:=5 RETURN
  LOOP
    cand:=k+7 RETURN
    cand:=k+11 RETURN
    cand:=k+13 RETURN
    cand:=k+17 RETURN
    cand:=k+19 RETURN
    cand:=k+23 RETURN
    cand:=k+29 RETURN
    cand:=(k:=k+30)+1 RETURN
  REPETITION
DONE(*suite_candidats*);

  positif VALUE premier_normal=SUCC premiers_speciaux;
  (*le rang du premier nombre premier dont il faut tester
    la divisibilite explicitement
  *)

  Boolean FUNCTION premier
    (positif VALUE candidat)
  (*vrai ssi  candidat  est un nombre premier*)
  DECLARE(*premier*)
    integer VARIABLE pos:=premier_normal
  DO(*premier*)TAKE
    CYCLE examen REPEAT

    IF
      premiers[pos]>candidat%premiers[pos]
    EXIT examen TAKE TRUE DONE;

    IF
      candidat\premiers[pos]=0
    EXIT examen TAKE FALSE DONE;

      pos:=SUCC pos
    REPETITION
  DONE(*premier*);

  ACTOR action VARIABLE insere_requete;
  (*variable introduite pour cause de recursion mutuelle*)

  PROCESS produit_premier
    ATTRIBUTE rang
    (positif VALUE rang)
  (*cherche et imprime le nombre premier de rang donne*)
  DO(*produit_premier*)RETURN
    UNLESS rang<=nombre_formes THEN
      print(line,"--->Requete"_,edit(rang,6,0),
                _"enregistree<---",line);
      insere_requete EVAL;
      print(line,___"PRIERE DE PATIENTER",line)
    RETURN DONE;
```

```
    print("Nombre premier de rang"_,edit(rang,6,0),_"="_,
          edit(premiers[rang],7,0),line)
  DONE(*produit_premier*);

  CLASS queue_requetes
    VALUE moi
    ATTRIBUTE vide,premier,enfiler,defiler
  (*Une queue de priorite, initialement vide, de coroutines
    du type  produit_premier  ordonnees selon les valeurs
    croissantes de leur  rang
  *)
  DECLARE(*queue_requetes*)
    OBJECT arbre
      (produit_premier VALUE requete;
       arbre VARIABLE gauche,droite)
     VARIABLE racine:=NIL;

    Boolean EXPRESSION vide=
    (*vrai ssi la queue est vide*)
      racine=NIL;

    produit_premier EXPRESSION premier=
    (*l'element prioritaire de la queue; si cette derniere
      est vide, le resultat est  NONE
    *)
      CONNECT racine THEN
        requete
      DEFAULT NONE DONE;

    arbre FUNCTION fusion
      (arbre VARIABLE p,q)
    DECLARE arbre REFERENCE r->p DO
      WITHIN q REPEAT
        IF TAKE
          UNLESS r=NIL THEN
            (*q.*)requete.rang<r.requete.rang
          DEFAULT TRUE DONE
        THEN r:=:q DONE;
        CONNECT r THEN
          r->gauche:=:droite
        DONE
      REPETITION
    TAKE p DONE(*fusion*);

    queue_requetes FUNCTION enfiler
      (produit_premier VALUE requ)
    (*Insere l'objet  requ  dans la queue; le resultat est la
      queue concernee
    *)
    DO racine:=fusion(racine,arbre(requ,NIL,NIL)) TAKE moi DONE;

    queue_requetes FUNCTION defiler
```

```
        (produit_premier REFERENCE requ)
      (*elimine de la queue son element prioritaire et le place
        dans la variable associee a  requ ; si la queue est
        vide, stocke NONE en  requ . Le resultat est la queue
        concernee
      *)
      DO(*defiler*)
        requ:=CONNECT racine THEN
                racine:=fusion(gauche,droite) TAKE requete
              DEFAULT NONE DONE
      TAKE moi DONE(*defiler*)
    DO(*queue_requetes*)DONE VALUE
      demandes_en_suspens=queue_requetes;

MODULE initialise_insere_requete DO
  insere_requete:=
    BODY action DO
      demandes_en_suspens.enfiler(produit_premier[CURRENT])
    DONE
DONE(*initialise_requete*);

COROUTINE traite_demandes_en_suspens DECLARE
  produit_premier VARIABLE requete
DO
  LOOP RETURN
    UNTIL demandes_en_suspens.vide REPEAT
      WHILE
        nombre_formes<demandes_en_suspens.premier.rang
      REPEAT
        UNTIL premier(prochain) REPETITION;
        insere_premier(courant)
      REPETITION;
      demandes_en_suspens.defiler(requete);
      ACTIVATE requete NOW
    REPETITION
  REPETITION
DONE(*traite_demandes_en_suspens*);

COROUTINE termine_session
  ATTRIBUTE fin_demandee
(*imprime la liste des nombre premiers obtenus et suspend
  l'execution de la paroutine  genere_premiers
*)
DECLARE(*termine_session*)
  Boolean VARIABLE fin VALUE fin_demandee:=FALSE
DO(*termine_session*)RETURN
  fin:=TRUE
DONE(*termine_session*)

DO(*genere_premiers*)
  (*initialise la table*)
  FOR integer FROM 1 TO premier_normal REPEAT
```

```
premiers                                        Vax Newton Compiler 0.2c16
                                            Page    5
                   Source listing

        insere_premier(prochain)
      REPETITION
      (*des mainteneant, on peut utiliser le systeme*)
    INTERRUPT(*l'utlisateur le relancera*)
      (*genere les autres nombres*)
      UNTIL
        ACTIVATE traite_demandes_en_suspens NOW
      TAKE fin_demandee REPEAT
        UNTIL premier(prochain) REPETITION;
        insere_premier(courant)
      REPETITION;
      print(line,"***Liste des nombres premiers obtenus***",line);
      FOR integer VALUE k FROM 1 TO nombre_formes REPEAT
        IF k\10=1 THEN line DONE; edit(premiers[k],7,0)
      REPETITION;
      print(line,"<<<FIN>>>")
    DONE(*genere_premiers*);
    /* /*EJECT*/ */
```

Par contre, *suite_candidats* et *traite_demandes_en_suspens* sont des coroutines normales.

La tâche *génère_premiers* produit continuellement les nombres premiers successifs dans la rangée extensible *premiers*; la variable *nformés* de valeur *nombre_formés* indique, à chaque moment, le nombre des nombres premiers qui ont été formés et enregistrés dans cette rangée. Le stockage dans la rangée a lieu par l'intermédiaire de la procédure *insère_premiers*; on remarque que l'on a pris soin d'insérer la valeur donnée *prem* dans cette rangée avant d'incrémenter la variable *nformés*. Ceci est important; l'assignation *premiers [(nformés :=* **succ** *nformés)] := prem* s'avèrerait incorrecte si un message parvenait entre le moment de l'incrémentation de la variable *nformés* et le stockage de la valeur correspondante *prem* dans la rangée.

La coroutine *suite_candidats* produit comme candidats à la primalité d'abord les *premiers_spéciaux = 3* valeurs *2*, *3* et *5*; ensuite, elle ne génère que les entiers qui ne sont multiples ni de *2*, ni de *3*, ni de *5*. Le test de si un candidat donné est premier est incorporé dans la fonction *premier* : il suffit pour celà d'examiner s'il n'est divisible par aucun des membres de la rangée *premiers* d'indice *pos* supérieur à *premiers_spéciaux* et dont le carré de la valeur *premiers [pos]* ne dépasse pas le candidat en question.

Une requête d'utilisateur est enregistrée sous la forme d'un message du type *produit_premier*. Les requêtes qui ne peuvent être satisfaites immédiatement sont insérées dans la queue de priorité *demandes_en_suspens* par l'intermédiaire de l'énoncé *insère_requête* **eval**; le recours à cet objet procédural, initialisé dans le module *initialise_insère_requête*, est lié à une récursion mutuelle entre le processus *produit_premier* et la classe *queue_requêtes*.

La coroutine *traite_demandes_en_suspens* est attachée, par la partie exécutable de la tâche *génère_premiers*, pour traiter les requêtes enregistrées dans la queue *demandes_en_suspens*. Pour celà, et tant que la queue n'est pas vide, elle produit les nombres successifs jusqu'à ce que la requête prioritaire *demandes_en_suspens.premier* puisse être satisfaite. Elle l'enlève alors de la queue et la réactive. Dès que la queue est vide, cette coroutine se détache.

La coroutine *termine_session*, mise en oeuvre par un message ad-hoc lorsque l'exécution du système doit être arrêtée, ne fait que changer la valeur *fin_demandée* de sa variable *fin* de **false** à **true**. C'est la partie exécutable de la tâche *génère_premiers* qui effectuera l'impression finale

de la table des nombres premiers obtenus dès que *fin_demandée* devient vraie et qu'il n'y a plus de demande en suspens.

On remarque la clause **interrupt** au début de la partie exécutable de la tâche *génère_premiers*. Des requêtes ne peuvent être enregistrées que lorsque la tâche aura été relancée à la suite de ce point d'arrêt : à ce moment, l'initialisation des structures de données requises aura été accomplie. Pour le reste, on constate qui si aucune demande n'est en suspens (et si la fin du traitement n'a pas été demandée), la tâche *génère_premiers* fait calculer un à un les nombres premiers suivants en examinant, entre chacun, si une requête n'a pas été enregistrée dans la queue *demandes_en_suspens* ou si la fin de la session n'a pas été demandée entre-temps.

La tâche incorporée dans la partie exécutable du programme *premiers* ne nécessite presque pas de commentaire. On peut simplement remarquer qu'au début de son exécution, elle attend que *génère_premiers* se soit interrompu après avoir initialisé les structures de données qui y sont incorporées et qu'elle relance cette tâche avant d'accepter les requêtes d'utilisateur.

```
premiers                                          Vax Newton Compiler 0.2c16
                               Page      6
                 Source listing


        positif VARIABLE rg
      DO(*premiers*)
        (*s'assure que l'on puisse utiliser la paroutine
          genere_premiers
        *)
        UNTIL STATE genere_premiers=waiting REPEAT SWITCH REPETITION;
        (*c'est bon, elle a atteint son premier point d'arret apres
          sa partie declarative
        *)
        SCHEDULE genere_premiers NOW;
        UNTIL end_file REPEAT
          UNTIL end_line REPEAT
            read(rg); produit_premier(rg) SEND
          REPETITION;
          next_char
        REPETITION;
        termine_session SEND
      DONE(*premiers*)

      **** No messages were issued ****
```

FIN DU COURS